Algovize

aneb

procházka krajinou

algoritmů

Luděk Kučera

Tato kniha slouží jako průvodce applety z **www.algovision.org**

Algovize
aneb procházka krajinou algoritmů
1. vydání, 2009

Luděk Kučera

Pro Univerzitu Karlovu v Praze vydala Blatenská tiskárna s.r.o.
Vytiskla Blatenská tiskárna s.r.o.

ISBN 978-80-902938-5-4

Vyšlo s podporou rozvojového programu Algovize (2004-2006)
Ministerstva školství, mládeže a tělovýchovy České republiky

Úvod

Algovize je soubor appletů, které názorným způsobem vykládají podstatu a činnost některých základních algoritmů, které se používají v informatice. Algovize vznikala postupně jako pomůcka pro přednášku Algoritmy a datové struktury na matematiko-fyzikální fakultě v Praze. Přednáška byly zavedena počátkem devadesátých let minulého století; pro větší názornost jsem při výkladu začal používat nejprve PowerPointové obrázky. Ty byly zpočátku statické, ale povaha předmětu si žádala o animace. Záhy se ukázalo, že PowerPoint je pro animace příliš primitivní a od té doby jsou vizualizace algoritmů implementovány jako applety napsané v jazyce Java. Je sice možné, že by v některých současných jiných jazycích a prostředích poskytujícími dynamické grafické prostředky mohly být vizualizace ještě lepší (především rychlejší), ale velkou výhodou Javy je, že běžné webové prohlížeče obvykle umožňují prohlížení appletů a tím je Algovize přístupna širokému okruhu zájemců - každému, kdo má přístup k Internetu.

Vizualizační applety ale nejsou jednoduché scénky. Každý z nich je komponován jako podpora přednášky o konkrétním algoritmu a proto se skládá z mnoha scén, každá z nich má specifický způsob obsluhy, vysvětluje mnohdy velmi obtížnou látku a proto vyžaduje i velmi obsáhlý návod k účelnému použití a popis všeho, co je s jeho pomocí vykládáno.

Původní koncepce vkládala vysvětlující texty do okének, které byly součástí grafické prezentace na obrazovce nebo promítací plátně. Ukázalo se ale, že textu bylo příliš mnoho a jeho vložení na obrazovku bylo na úkor přehlednosti a srozumitelnosti presentace. Proto jsem se nakonec rozhodl nechat na obrazovce jen pohyblivé obrázky a text přenést do knihy, které bude uživateli průvodcem po jednotlivých vizualizacích.

Kniha je rozdělena do sedmi částí, z nichž každá odpovídá ucelené oblasti informatiky: datové struktury, třídění posloupností, grafové algoritmy, aritmetické algoritmy, výpočetní geometrie, vyhledávání v textu a optimalizace. Každá část se pak dále dělí na kapitoly, které odpovídají klasickým výpočetním problémům. Pokud je pro jeden problém (například hledání nejkratší cesty) uvedeno více alternativních algoritmů, je ještě kapitola rozdělena na

podkapitoly. Text je současně návod k použití příslušného appletu Algovize a také vysvětlením myšlenky, na které je daný algoritmus založen, intuitivním způsobem za pomoci pohyblivých obrázků.

Text provázející appletem stejně tak jako applet sám jsou rozděleny do scén, které si navzájem odpovídají a to i jménem. Je vhodné číst knihu současně s prohlížením appletu, nebo lépe řečeno, při procházce appletem je vhodné se řídit návodem průvodce. Scény představují doporučený způsob prohlídky appletu, které sledují způsob, jakým je odpovídající algoritmus vykládán na přednášce, ze které Algovize vznikla.

Tato kniha *není* učebnicí algoritmů. Neuvidíte zápis vykládaného algoritmu formálním způsobem (pseudokód), nebude dokázáno, že algorithmus funguje správně a že výpočet někdy skončí, velmi málo se dozvíte o motivaci nebo historii problému a neuvádím citace z literatury. Mojí snahou je, abyste ale co nejrychleji a nejhlouběji pochopili, jaká myšlenka algoritmus pohání a proč se vše dělá tak, jak se to dělá. Míníte-li své studium algoritmů vážně, budete si muset opravdovou učebnici půjčit nebo koupit a vše ještě důkladně prostudovat, ale doufám, že s pomocí Algovize vám vše půjde daleko rychleji, než kdybyste se do studia klasické učebnice pustili rovnou.

Módním trendem je nabízet výuku, která má být zábavná. I když předkládané applety jsou založeny na sledování pohybující se obrázků na obrazovce nebo promítacím plátně, takže by je bylo možno nabízet pod heslem "learning is fun", *nejsou* koncipovány jako zábava ale jako pomůcka, která uživatele přiměje k usilovné pozornosti v přednáškové místnosti a k tvrdé práci doma. Nejsou ani koncipovány jako prostá animace, která graficky a v čase ukazuje, jak se mění hodnoty proměnných algoritmu. Slovo "animace" nechápu na rozdíl od většiny svých kolegů jako odvozené od slova "animal", které v angličtině i francouzštině znamená zvíře, tvora který se sám hýbe, ale není nadán vyšší inteligencí. Animace pochází od latinského slova "anima" - duše. Mojí snahou je vytvářet inteligentní applety s duší, které se snaží vizualizovat především podstatu toho, co se při výpočtu děje a proč se to děje, neboli ukázat myšlenku, na které tvůrce algoritmu svůj postup založil. Mělo by být snahou uživatele tuto duši nalézt.

Tvůrčí osobnost má intuici, představu ze které vše přirozeně vyplývá. Alespoň u sebe někdy pozoruji, že se intuice projevuje před vnitřním zrakem jako obrázky a právě tyto obrázky a jejich dynamiku se snažím přenést na obrazovku. Zda jsem byl ve své snaze převést neviditelné na viditelné úspěšný nechť posoudí laskavý čtenář mé knihy a současně divák mých appletů. Pokud by se můj záměr podařil, mé applety by měly přinést nejen poučení, ale také *potěšení* pochopit myšlenky nejlepších informatiků posledních padesáti let, což je daleko více než pouhá zábava.

Luděk Kučera, Praha, srpen 2009

Část I

Datové struktury

V první kapitole této knihy se budeme zabývat způsoby, jak je možno efektivním způsobem ukládat informaci o posloupnosti a množině. Znalost těchto metod patří mezi základní programátorské dovednosti.

Posloupnost je soubor prvků, obvykle čísel, u kterého záleží nejen na tom, které prvky do ní patří, ale také v jakém pořadí se v posloupnosti nacházejí. Připouští se, aby týž prvek (číslo) se v posloupnosti vyskytoval vícekrát. Budeme se zabývat tím, jak dlouho trvá provedení základních operací jako je přidání prvku do určeného místa v posloupnosti, vynechání prvku z posloupnosti, zjištění, zda a popřípadě kde daný prvek v posloupnosti leží, určení předchůdce a následníka prvku a pod. Těmito otázkami se zabývá první kapitola této části, kde posloupnosti nazýváme po programátorsku seznamy.

Množina je soubor prvků, obvykle čísel, u kterého pořadí prvků není určováno. Předpokládá se, že libovolný prvek (číslo) se v množině vyskytuje jen jednou. Budeme se zabývat tím, jak dlouho trvá provedení základních operací jako je přidání prvku do množiny, vynechání prvku z množiny, zjištění, zda daný prvek v množině leží, nalezení minimálního a maximálního prvku a pod. Těmito otázkami se zabývá druhá a třetí kapitola této části. Na rozdíl od první kapitoly, kde implementace seznamů je přímočará a jednoduchá, struktury pro reprezentaci množin, které uvedeme v druhé a třetí kapitole, budou značně náročné, ale jejich výhodou je velmi rychlé provádění všech základních operací i pro velké množiny.

Halda (anglicky. heap) je vlastně speciální způsob implementace množinové datové struktury, kde nás nezajímá zjišťování, zda daný prvek v množině leží. Základní množinové operace, které halda umožňuje efektivně provádět je přidání prvku a nalezení a vynechání minimálního prvku. Původní aplikací haldy bylo třídění posloupnosti (algoritmus Heapsort), ale pak se ukázalo, že halda má samostatné použití v řadě jiných algoritmů. Ve čtvrté kapitole této části ukážeme, že omezení souboru povolených operací umožní haldu efektivně implementovat způsoby podstatně jednoduššími než jsou množinové implementace z předchozích dvou kapitol.

Slučovatelná halda je implementace haldy, která umožní kromě výše uvedených operací provádět i sjednocení (sloučení) dvou různých disjunktních hald, což je u některých algoritmů užitečné. Popsaná elegantní implementace binomickou haldou je založena na velmi zajímavém nápadu.

Konečně v poslední kapitole bude popsána datové struktura odlišného charakteru, která umožňuje popsat způsob rozložení dané množiny do navzájem disjunktních tříd a provádění několika základních operací jako je určení, do které třídy patří daný prvek množiny nebo sloučit dvě třídy rozkladu. V této knize je tato datová struktura použita při hledání minimální kostry grafu, ale je uvedena odděleně, protože má i další aplikace.

Kapitola 1

Seznamy

Jednou ze základních datových struktur je *seznam*. Dobrá znalost různých implementací seznamů patří k základním dovednostem každého programátora.

Seznam je vlastně posloupnost objektů jistého typu; v našich appletech to budou celá čísla, ale obecně to mohou být i velmi složité objekty nejrůznějšího druhu. Objekty bývají nějak označeny nebo pojmenovány číslem, které budeme nazývat *klíč*. Lze tedy říci, že naše applety objekty redukují na samotný klíč, protože to je to jediné, co je návrháři seznamu z objektu přístupné a viditelné.

V této kapitole se budeme zabývat jednak tím, jak seznam uložit v paměti počítače, ale především jak se seznamy provádět nejrůznější operace. Uveďme hlavní operace:

- **Inicializace** je sice nutnou, ale obvykle velmi jednoduchou operací na začátek, která vytvoří prázdný seznam, do kterého je pak možno přidávat.

- **Hledání** je nejčastěji používanou operací a jejím cílem je zjistit, zda v seznamu je objekt s daným klíčem. Jelikož se připouští, aby v seznamu mohlo být více objektů se stejným klíčem, má hledání řadu variant: nalezení prvního, posledního nebo libovolného objektu s hledaným klíčem nebo určení všech takových objektů.

- **Vkládání** je další důležitou operací. Po inicializaci je obvykle seznam prázdný a je nutné jej naplnit a často se přidávání provádí i průběžně během celého života daného seznamu. Přidáváme-li do seznamu nový objekt s jistým klíčem, je třeba určit, kam má být vložen a i zde je mnoho variant: na začátek, na konec, před nebo za zadaný prvek seznamu nebo do libovolného místa.

- **Vynechávání** je také přirozenou operací. Obvykle je dán prvek seznamu (například pomocí ukazatele, který na něj odkazuje) a tento prvek je třeba ze seznamu vypustit. Existují ale různé obměny zadávání prvku určeného k vynechání, například vynechat první nebo poslední prvek seznamu nebo prvek, který se nachází před nebo za zadaným prvkem seznamu. Uvidíme například, že v jednoduše zřetězeném seznamu je obtížné vynechat zvolený prvek, ale velmi rychle se dá vynechat prvek, který se nachází za ním.

- **Určení minima nebo maxima** znamená obvykle projít celý seznam a poznamenat si jeho prvek s nejmenším nebo největším klíčem (a pokud může takových extrémních prvků být více, pak zase máme varianty jako u vyhledávání: nalezení prvního, posledního nebo libovolného minima/maxima nebo všech prvků s extrémní hodnotou klíče).

- **Určení prvního nebo posledního prvku** je operace, jejíž náročnost je velmi závislá na způsobu implementace seznamu. Někdy je triviální - příslušnou informaci máme stále explicitně poznamenanou; jindy je zase zdlouhavá (třeba když máme poznamenáno pouze kde seznam začíná a k nalezení jeho konce musíme celým seznamem projít).

- **Určení následníka nebo předchůdce daného prvku** je operace, která je užitečná sama o sobě, ale ještě častěji se používá jako součást některé z předchozích operací. Například vynechání prvku zřetězeného seznamu se provede tak, že se přímo propojí jeho předchůdce a jeho následník a tím prvek samotný ze seznamu vypadne. Jedná se o operaci, jejíž časová náročnost je velmi závislá na implementaci seznamu; někdy je tato informace explicitně poznamenána v objektu, představujícím prvek, jindy je například určení předchůdce velmi zdlouhavé.

- **Určení délky seznamu**; pokud tuto operaci potřebujeme, je výhodné mít tuto informaci explicitně poznamenánu a hodnotu inkrementovat při přidávání a dekrementovat při vynechávání. V následujícím se jí proto přímo zabývat nebudeme, i když je mnohdy velmi potřebná.

- **Rozštěpení seznamu na dva**: zadáme jeden prvek seznamu a úkolem je rozdělit seznam na dva, přičemž prvý jde od začátku původního seznamu po zvolený prvek a druhý je tvořen prvky, které následují za zvoleným prvkem. Obvykle se spokojíme s tím, že přitom původní seznam zanikne.

- **Sloučení dvou seznamů do jednoho**: ze dvou seznamů se vytvoří jeden tak, že za poslední prvek prvního seznamu připojíme první prvek druhého seznamu. Obvykle se spokojíme s tím, že přitom původní seznamy zaniknou.

Běžné implementace seznamů se dají rozdělit do dvou tříd: kompaktní seznamy (též popisované jako seznamy v poli) a zřetězené seznamy. Zřetězené seznamy se zase dělí na jednoduše a obousměrně zřetězené. Každá z těchto tří tříd má mnoho různých variant, z nichž zde probereme jen některé a existuje i mnoho implementací, které do naší klasifikace úplně nezapadají. Přesto však se čtenář v této kapitole dozví základní myšlenky a techniky, které se při práci se seznamy používají a jistě mu nebude činit obtíže pochopit i různé varianty, které zde nejsou probrány.

1.1 Kompaktní seznamy

Pro ukládání informace formou kompaktního seznamu se vytvoří pole, jehož jednotlivé položky mohou obsahovat objekty, představující jeho prvky. Jestliže tedy prvky seznamu budou objekty třídy *Obj*, pak takové pole A se deklaruje
v Pascalu jako A: `array`$[1..N]$ `of` *Obj*
v C a odvozených jazycích jako *Obj*$[N]$ A
pro nějaké číslo N.

Prvky seznamu se nejčastěji ukládají do pole "zleva doprava", neboli seznam obsahující n prvků ($n < N$) se při popisu v C a odvozených jazycích uloží do $A[0], A[1], \ldots, A[n-1]$ a při popisu v Pascalu do $A[1], A[2], \ldots, A[n]$. V appletech budeme položky pole znázorňovat jako malé čtverečky; ty které neobsahují užitečnou informaci (neboli neobsahují prvky seznamu) budou šedivé a "aktivní" položky budou mít jasnou barvu (například zelenou).

Nevýhodou kompaktního seznamu je to, že alokace paměti pro uložení pole se provádí na začátku při inicializaci seznamu nebo automaticky před začátkem výpočtu a velikost pole (tedy hodnota N z uvedených deklarací) musí být alespoň tak velká jako maximum délky seznamu v průběhu celého výpočtu. Někdy tuto hodnotu neznáme a může dojít k přetečení pole. Jindy jsme příliš opatrní a průměrná délka seznamu je výrazně kratší než jeho maximumální délka a proto používáme paměť počítače neefektivně - velká část pole většinou neobsahuje žádnou užitečnou informaci, ale zabírá paměť, která byla programu alokována.

Alokace pole se proto v některých případech v průběhu výpočtu mění: dojde-li k přetečení, alokuje se nové a větší pole, seznam se do něho překopíruje a původně alokovaná paměť se uvolní (vrátí explicitně nebo implicitně operačnímu systému počítače). Pokud je naopak seznam mnohem kratší, alokuje se nové, kratší pole, seznam se zkopíruje a původní velký a nevyužívaný segment paměti se uvolní. Tím se dosáhne účelného využívání paměti a nehrozí nebezpečí kolapsu výpočtu přetečením, ale alokace a dealokace paměti i přenášení seznamu z místa na místo nepůsobí příznivě na rychlost výpočtu. Můj první počítač Sinclair ZX81, který jsem si koupil v roce 1982, měl 1 kB operační paměti RAM a proto pro úsporu místa používal výhradně optimálně

alokované kompaktní seznamy a opravdu hodně času strávil jejich přenášením sem a tam.

V našich scénách se problémy s volbou velikosti pole a jeho alokací zabývat nebudeme.

Výhodou kompaktních seznamů je jejich jednoduchost i snadné určení předchůdce a následníka (sousedi v poli), ale na druhé straně při vkládání a vynechávání se mnoho prvků pole musí neustále v poli přesouvat sem a tam. Je proto nutné se zamyslet nad tím zda kompaktní seznam je vhodný pro zamýšlenou aplikaci - někdy to může být ideální volba, při jiných požadavcích na prováděné operace může být zcela nevhodný.

Scéna: *Operace s kompaktním seznamem*

Na rozdíl od algoritmů v dalších částech knihy zde ukážeme všechny operace s kompaktním seznamem najednou, protože jsou velmi jednoduché a přirozené a jejich pochopení nepřináší zvláštní obtíže. Popis scény se v podstatě omezí na popis práce s ovladačem.

Knoflíkem **Prázdný** se seznam vyprázdní a knoflíkem **Nový** se vytvoří nové pole pro seznam, které má délku udanou v poli **N=,** obsahující tolik náhodně vygenerovaných prvků, kolik je zapsáno v poli vedle $n =$**.** Tyto ovládací prvky použijete pro obvykle pro vytvoření výchozího seznamu.

Ovladač zahrnuje volbu operace, která má být provedena.

Při vyhledávání je vyhledávaný klíč určen hodnotou v poli **Klíč,** která je pro každou operaci znovu náhodně nastavena, ale je možno ji přepsat. Pak je nutné zvolit variantu vyhledávání. Jistě vám nepřijde divné, že vyhledávání prvního výskytu (t.j. výskytu nejbližšího levému konci pole) se v appletu provádí zleva, hledání posledního výskytu začíná zprava. U ostatních variant je to lhostejné, zde je zvolena varianta zleva. Postup hledání je znázorněn šipkou, pohybující se pod polem, ukazující na aktivní položku pole. Při nalezení hledaného klíče příslušné čtvereček zčervená.

Při přidávání je třeba specifikovat místo, kam má přidávaný prvek vložen; na ovladači se objeví příslušná volba. Pokud se zvolí **Před** nebo **Za** (rozumí se před nebo za zavolený prvek), je třeba myší zvolit prvek seznamu klepnutím myší (zčervená). Volba **Libovolně** je zde ekvivalentní volbě **Na konec,** protože, jak jistě poznáte, přidávání prvku na konec seznamu je nejjednodušší a proto je-li nám jedno, kam prvek přidat, je vhodné ho dát na konec. Klíč nového prvku seznamu je určen hodnotou v pole vedle **Klíč,** která je pro každou operaci znovu náhodně nastavena, ale je možno ji přepsat.

Pokud přidávaný prvek nepřijde nakonec, musíme mu v seznamu vytvořit volné místo tím, že všechny prvky v seznamu od zvoleného místa vpravo (včetně) se posunou o jednu položku doprava. Tento posun je přitom nutné provádět od pravého konce přesouvané oblasti, aby nedošlo k přemazávání dosud stojícího prvku přesouvaným.

Při vynechávání se také objeví volba variant. Při volbě **Zvolený, Před** nebo **Za** (tedy vynechávání zvoleného prvku nebo prvku před ním nebo za ním) je nutné klepnutím myši zvolit prvek seznamu, udávající, kde k vynechání dojde.

Prvek se ze seznamu vynechá a pokud nebyl na pravém konci seznamu, pak se musí prázdné místo zaplnit prvky pole, které leží vpravo od prázdného místa. Prvky se přesunou o jednu pozici vlevo a přesouvání zde musí začít (na rozdíl od přidávání) vlevo, aby se zabránilo vzájemnému přemazávání prvků seznamu.

Hledání minima nebo maxima musí projít celý seznam (zde zleva doprava); průběžný extrém je zaznamenáván v boxu nad polem, kde pak na konci zůstane celkový výsledek operace.

U kompaktního seznamu je nutné mít explicitně poznamenanou dyyyyélku seznamu, aby bylo zřejmé, která část pole popisuje seznam (tato hodnota je uvedena ve žlutém boxu nad polem). Pak je ale triviální určit první a poslední prvek seznamu. Stejně triviální je určit předchůdce nebo následníka prvku se známou polohou (nebo zjištění, že neexistuje). Tyto operace proto v appletu ukázány nejsou.

Při rozdělování seznamu do dvou se musí všechny prvky druhého seznamu překopírovat do jiného pole, které buď dostaneme nebo musíme vytvořit pro přijetí oddělovaného seznamu. Podobně se při slučování seznamu druhý ze seznamů musí překopírovat do pole, ve kterém je uložen první seznam, a to bezprostředně za jeho poslední prvek. Obě operace tedy mohou zahrnovat přenosy velkého množství prvků a jsou proto obecně časově náročné.

Prováděním operací dojdete k poznatku, že pokud je seznam hodně dlouhý, provádění operací je většinou pomalé. Buď projdeme celý seznam (minimum, maximum, neúspěšné hledání) nebo celý seznam přesouváme (přidávání a ubírání na začátku) a nebo operujeme s obecně velkou částí seznamu, v průměru zhruba polovinou (úspěšné hledání, přidávání nebo ubírání v obecné poloze).

Nakonec deaktivujte volbu **Ukaž všechny.** Ukázána je nyní jen hodnota v aktivním poli, které je ukazováno šipkou. Takto kompaktní seznam "vidí" počítač (přesněji jeho procesor). Zkuste si všechny operace ještě jednou; u některých získáte přesnější představu o povaze operace. Například při vyhledávání lidské oko stihne prohlédnout seznam s vyplněnými hodnotami najednou a určit výsledek rychle, zatímco při slepém prohledávání lépe pochopíte sekvenčnost celé operace.

Scéna: *Zásobník*

Tato scéna ukazuje zásobník, implementovaný jako kompaktní seznam. Jedná se o datovou strukturu, označovanou také jako LIFO (z anglického last-in-first-out neboli poslední dovnitř, první ven), která umožňuje jen přidávání a vynechávání na konci a která funguje jako stoh košil v prádelníku: vyprané a

vyžehlené košile přidáváme nahoru a horní košili si také zase vezmeme, protože by se jinak pěkně srovnaný sloupec rozházel.

Sice by se mohlo zdát, že takto omezená datová struktura není příliš užitečná, ale možná již víte, že pravý opak je pravdou - zásobník má velmi široké použití v mnoha úlohách, například se osvědčuje pro popis niti, kterou odvíjíme při prohledávání bludiště, abychom trefili zpět k vstupu.

Zásobník se dá výborně implementovat kompaktním seznamem, protože využívá jediné dvě varianty přidávání a vynechávání, které jsou provedeny bleskově i pro velké seznamy, totiž přidávání a vynechávání na (pravém) konci seznamu, které nevyžadují žádné přesouvání prvků.

Jedinou potíží je, že velikost pole pro zásobník je třeba znát dopředu, a pokud je maximální počet prvků uložených v zásobníku výrazně menší než průměrný počet, pak je pole většinu doby poloprázné a tedy plýtváme pamětí.

Chvíli si zkuste se zásobníkem pohrát pomocí knoflíků **Vsuň** (klíč zapsaný v poli vedle **Klíč)** a **Vyjmi,** ale jistě vás to brzo omrzí, protože je to příliš lehké. Zásobník je ukázán jen proto, že je to důležitá datová struktura a ne proto, že by byl sám o sobě zajímavý.

Scéna: *Fronta*

Kromě zásobníku je také často používanou jednoduchou datovou strukturou *fronta*, někdy označována jako FIFO fronta (z anglického first-in-first-out, neboli první dovnitř, první ven), kde se jako ve frontě u obchodu příchozí řadí na konec a pak se ve frontě posouvají pomalu dopředu, aby na jejím začátku byli obslouženi (a opustili ji). Pokud nový prvek seznamu přidáváme na jeho (pravý) konec, jde to provést rychle. Při vynechání prvku, který je na začátku (tedy na levém konci fronty) bychom měli všechny zbylé prvky seznamu přesunout o jednu pozici doleva. Pro úsporu času to ale neuděláme.

Popsaná úprava má ovšem za následek, že se fronta postupně posunuje vpravo a pracujeme-li s ní déle, její konec narazí na pravý okraj pole. Problém s přidáním dalšího prvku vyřešíme tím, že fronta bude pokračovat v nyní volném prostoru od začátku fronty. Je to tedy, jako bychom pole "zavinuli" do kruhu - spojili pravý konec pole s jeho levým koncem. Zvolte možnost **Zavinutá** namísto **Lineární** a s malou animací se ukáže tato zavinutá reprezentace. Pokračujíce v lineárním znázornění, vidíme, že fronta pokračuje skokem přes pravý okraj pole na jeho levý okraj. Pokračujeme-li dále, pravý segment fronty se zmenšuje, až po čase zmizí úplně a fronta se znovu objeví na začátku pole.

Zde je pochopitelně nutné si pamatovat nejen jak je seznam dlouhý (nebo kde končí, což v minulé scéně bylo totéž, ale nyní již ne), ale také kde začíná.

Pro programátory v jazycích vycházejících z C, kteří ukládají frontu do pole s položkami očíslovanými jako $A[0], A[1], \ldots, A[N-1]$, se doporučuje používat dvě proměnné: B ukazující na začátek fronty a E ukazující na první

nevyužitou položku pole ležící vpravo od položky zaujímané koncem fronty (s jednou výjimkou - pokud pravý konec fronty je shodný s pravým koncem pole, pak položíme $E = N$). Pokud je fronta obsahující n prvků v základní poloze, opřená o levý konec pole, pak je $B = 0$, $E = n$ a fronta je uložena v položkách $A[0], A[1], \ldots, A[n-1]$. Posune-li se fronta doprava, ale nepřeskakuje z pravého konce pole na levý, pak je $B \leq E$ (přičemž rovnost by znamenala, že fronta je prázdná - neobsahuje žádný prvek) a fronta je uložena v položkách $A[B], A[B+1], \ldots, A[E-1]$. Pokud je naopak $E < B$, znamená to, že fronta přeskakuje z pravého konce pole na jeho levý konec a je uložena v položkách $A[B], A[B+1], \ldots, A[N-1], A[0], \ldots, A[E-1]$. V appletu je pozice B ukazována modrou šipkou pod polem a E ukazuje zelená šipka (která je modrou šipkou zakryta, pokud je fronta prázdná, tedy $B = E$.

Scéna: *Oboustranná fronta*

Zajímavé je, že s využitím implementace z předchozí scény je možno jednoduše a rychle přidávat i na začátek fronty a vynechávat na jejím konci; tyto operace zase posouvají frontu vlevo a občas ji donutí přeskočit z levého okraje pole na pravý. Je to, jako bychom operace s FIFO frontou pouštěli pozpátku. Taková fronta se někdy označuje DEQUE (z anglického double-ended queue - fronta s oběma konci).

Použitím čtyř ovládacích knoflíků pro přidávání a vynechávání na levém a pravém konci fronty si operace zkuste. Jistě vidíte, že bez ohledu na velikost fronty je doba nutná k provedení libovolné z operací konstantní a velmi krátká.

Scéna: *Uspořádaný kompaktní seznam*

Velmi speciálním případem seznamu je uspořádaný seznam. Předpokládáme, že klíče je možno porovnávat podle velikosti (což je splněno například pokud klíče jsou celá čísla nebo obecněji reálná čísla). Uspořádaný seznam je takový, ve kterém jeho prvky představují neklesající posloupnost.

Tato scéna ukazuje uspořádaný seznam implementovaný kompaktním způsobem, protože na rozdíl od neuspořádaných seznamů (a také na rozdíl od zřetězených seznamů, které budou probírány dále) se dá velmi efektivně provádět vyhledávání prvku (tedy určení zda dané číslo je klíčem některého z prvků seznamu a popřípadě lokalizace takového prvku).

Sledujte vyhledávání na obrazovce: algoritmus pracuje tak, že velmi rychle zužuje oblast, ve které by se mohl hledaný klíč nacházet. Na počátku vycházíme z toho, že se hledaný klíč může nacházet kdekoli. Pokud má seznam n prvků, pak může být kdekoli mezi polohami 1 a n včetně. Levá a pravá hranice oblasti, ve které se hledaný klíč může nacházet, jsou na obrazovce označeny modrou a zelenou šipkou, které tedy na začátku ukazují na levý, respektive pravý konec seznamu.

Hledání spočívá v opakování následujícího kroku: jestliže jsme určili, že hledaný klíč může být mezi polohami i a j včetně, kde $i \leq j$, pak pokud $i = j$ nebo $i + 1 = j$, stačí se podívat na klíč i-tého prvku a (v druhém případě ještě j-tého prvku) a víme, zda klíč v seznamu skutečně leží (a kde) nebo zda se v seznamu nenachází. Je-li $i + 2 \leq j$, pak si jako m určíme prvek někde mezi i a j (výhodně $(i + j)/2$ popřípadě zaokrouhlené nahoru nebo dolu). Pak označíme-li jako k_m klíč m-tého prvku seznamu a k hledaný klíč, jsou tři možnosti:
$k < k_m$: pak hledaný klíč k musí ležet *vlevo* od zvoleného m-tého prvku, tedy horní mez j pro jeho výskyt lze snížit na $m - 1$;
$k = k_m$: pak byl hledaný klíč nalezen u m-tého prvku seznamu; a
$k_m < k$: pak hledaný klíč k musí ležet *vpravo* od zvoleného m-tého prvku, tedy dolní mez i pro jeho výskyt lze zvýšit na $m + 1$.

Šířka oblasti, ve které může hledaný klíč ležet, je proto na začátku n a dokud nepoklesne na 1 nebo 2, každým krokem se zmenší na polovinu. Počet opakování cyklu (který je sám o sobě velmi jednoduchý) je proto nejvýše $\lfloor \log_2 n \rfloor$.

Operace určování prvního a posledního prvku, což je v tomto případě totožné s určováním minima a maxima, stejně tak jako určování předchůdce a následníka prvku je podobně jako v obecných kompaktních seznamech okamžitá bez ohledu na velikost seznamu. Vynechávání prvku zase může být podobně jako v obecných kompaktních seznamech časově náročné, neboť zahrnuje přesun v nejhorším případě všech prvků, které v seznamu zbudou, o jednu polohu vlevo.

Přidávání do uspořádaného kompaktního seznamu se odlišuje od přidávání do obecného seznamu tím, že pokud v seznamu žádné dva různé prvky nemají stejné klíče, pak místo, kam přidávaný prvek má být přesunut, je jednoznačně určeno - musíme ho přidat tam, kde nebude porušovat uspořádání. I když připouštíme vícenásobný výskyt klíče, je místo vložení obvykle velmi silně omezeno. Stejné jako v obecných seznamech je to, že přidávání je i v uspořádaném kompaktním seznamu obecně neefektivní - například přidáváme-li prvek s klíčem menším než dosavadní minimum klíčů, musíme jej vložit na levý konec seznamu a to si vynutí přesun všech původních prvků seznamu o jednu pozici doprava.

Všechny dosud vyjmenované operace můžete v této scéně provádět, ale nedozvíte se přitom nic nového, co by již nebylo známo z předešlých scén. Velmi zajímavé a nové je ale u uspořádaného kompaktního seznamu vyhledávání.

Zatímco tedy se všechny dosavadní operace s kompaktními seznamy buďto prováděly velmi rychle (s konstantním a malým počtem kroků nezávislých na velikosti seznamu) a nebo naopak obecně velmi pomalu (což znamená, že bylo nutné manipulovat s podstatnou částí prvků - někdy dokonce se všemi), vyhledávání v uspořádaném kompaktním seznamu se provádí v logaritmickém

čase (tedy v čase úměrném $\log N$), který sice není konstantní, ale je velmi malý i pro velké počty prvků (např. pro jednu miliardu je $\log_2 N$ menší než 30).

Vyhledávání v uspořádaném seznamu je obdobou hledání, které člověk provádí v tištěném slovníku (nebo hledání dané stránky v libovolné knize.

1.2 Zřetězené seznamy

Kompaktní seznamy byly sice jednoduché, ale obvykle dosti pomalé. Jelikož jsme nepřipouštěli, aby v nich byly "díry", přidávání nebo vynechávání někde uprostřed vyžadovalo přesouvání mnoha prvků seznamu a bylo tedy zdlouhavé. Vyhledávání také znamenalo projít celý seznam nebo jeho značnou část, pokud jsme náhodou na hledaný klíč nenarazili hned zpočátku.

Zřetězené seznamy odstraňují první nevýhodu: prvky spolu nemusí v paměti počítače sousedit a proto nelze hovořit o dírách a naopak přidávaný prvek lze umístit libovolně a proto prvky seznamu mohou setrvávat na svých místech. Druhá nevýhoda - nutnost procházet celým seznamem při vyhledávání - je z principiálních důvodů společná všem seznamům a mají ji tedy i zřetězené seznamy.

Scéna: *Zřetězený seznam a jeho prohledávání*

U kompaktních seznamů bylo pořadí prvků v seznamu dáno polohou v poli. U zřetězených seznamů mohou být prvky rozmístěny v paměti počítače zcela libovolně a proto se jejich pořadí musí udávat explicitním způsobem. U (jednoduše) zřetězeného seznamu každý prvek obsahuje dva povinné údaje. Předně je to jeho klíč, jako tomu je u všech datových struktur, které v této knize budou probírány. Podle klíče prvek identifikujeme a vyhledáváme. Druhým údajem je *ukazatel* na následující prvek v seznamu. Seznam je potom popsán proměnnou, kterou budeme nazývat *záhlaví* seznamu; je to ukazatel na první prvek seznamu.

Ve scéně je znázorněn zřetězený seznam. Jeho záhlaví je představováno zeleným boxem v levé horní části obrazovky. Prvky seznamu jsou tvořeny obdélníky, které jsou zcela náhodně rozhozeny po ploše. Obdélníky jsou rozděleny na dvě části. Pravá obsahuje klíč tohoto prvku a v levé je uložen ukazatel na následující prvek.

Obvyklé vizualizace zřetězených seznamů ukazují klíče všech prvků jako čísla a ukazatele jako šipky. I zde si tyto informace můžete zobrazit zvolením checkboxu **Ukaž vše.** Nechte ale zatím tuto volbu neaktivní.

Informaci uloženou v prvcích seznamu jsem skryl záměrně, abych vám přiblížil situaci, ve které se nachází procesor počítače při procházení seznamem. Pokud totiž jsou zobrazeny všechny ukazatele, lidský pozorovatel obrázku má (není-li situace příliš nepřehledná) globální přehled po seznamu a mnohé, co ve skutečnosti je pro počítač časově náročné, vidí ihned.

Procesor ale "vidí", jen několik málo údajů, které si stáhne z paměti a chce-li znát další, pak jiné musí smazat. Zde se (pokud není zvoleno **Ukaž vše)** ukáže hodnota v záhlaví nebo ukazatel prvku seznamu pouze klepnete-li na příslušný box myší. Zobrazená informace zmizí, pokud klepnete na jinam. Vidět je tedy jen informace o jednom prvku.

Zkuste si nyní projít celým seznamem v tom pořadí, ve kterém po sobě v seznamu následují. Znamená to tedy klepnout na záhlaví pro informaci o poloze prvního prvku, pak na první prvek pro informaci o druhém prvku atd. Až se dostanete do prvku seznamu, který v boxu má černé kolečko představující nulový ukazatel, budete na konci seznamu.

Procházení seznamem lze také provádět automaticky přepnutím volby na ovladači z **Ručně** do polohy **Krokování** nebo **Animace.** V prvním případě Algovize odkrývá postupně jednotlivé prvky seznamu po kliknutí na knoflík **Krok,** v druhém případě stisknutí knoflíku **Animuj** spustí prohlídku seznamu jako souvislou animaci.

Prozradím vám, že na obrazovce jsou i některé obdélníky, které nejsou zapojeny do seznamu (ve skutečnosti patří do jiného seznamu, jehož použití uvidíme později). Klepnutím pravým knoflíkem myši si vyberte některý z obdélníčků na obrazovce (zmodrá), a zkuste zjistit, zda patří do seznamu, do kterého se vstupuje ze zeleného záhlaví, znázorněného v levém horním rohu obrazovky. Znamená to procházet seznamem, jak jsme to před chvílí dělali, tak dlouho, dokud na vybraný prvek s modrým obdélníčkem nenarazíme (pokud v seznamu leží) nebo dokud nedojdeme na konec seznamu (pokud v něm modrý obdélníček neleží). To je většinou velmi zdlouhavé.

Scéna: *Vyhledávání ve zřetězeném seznamu*

V minulé scéně jste se naučili procházet zřetězeným seznamem. Nyní si to zkusíme ještě jednou, ale s cílem vyhledat v seznamu zvolený klíč. Vyhledáváme hodnotu, která je na ovladači v poli vedle **Klíč.** Procházejte seznamem tak, jak jsme to dělali v minulé scéně; pokud je klíč nalezen, příslušný obdélník zčervená.

Opět je možno volit hledání prvního, libovolného, posledního nebo všech výskytů. V prvních dvou případech se můžeme zastavit, jakmile poprvé narazíme na hledaný klíč. Při hledání posledního výskytu ale nelze procházet seznam od konce a proto dokud nedojdeme až do posledního prvku seznamu, nemáme jistotu, zda případně nalezený prvek je *posledním* výskytem hledaného klíče. Stejně tak je nutné projít celým seznamem při hledání všech výskytů.

Pokud byste například hledali libovolný výskyt a náhodným klepnutím myši byste objevili obdélník, ve kterém klíč leží, nejásejte. Nemusí to být správná odpověď, protože, jak bylo řečeno v předchozí scéně, tento obdélník nemusí patřit do seznamu, ve kterém máte hledat a který je dostupný ze zeleného záhlaví v levé horní části obrazovky, protože na ploše jsou záměrně

pro zkomplikování situace znázorněny i prvky jiného seznamu.

Opět je možno zvolit si ruční průchod seznamem, krokování nebo animaci. Ruční průchod je chápán jako test porozumění výkladu. V okamžiku, kdy znáte odpověď (tedy ani dříve ani později) stiskněte ten z knoflíků **Nalezen** a **Nenalezen.** Dozvíte se, zda jste odpověděli správně. Pamatujte, že například při hledání posledního výskytu nebo všech výskytů je hotovo až na konci seznamu, protože v posledním prvku seznamu se může ukrývat poslední z výskytů hledaného klíče.

Scéna: *Předchůdce ve zřetězeném seznamu*

Než se pustíme do operací přidávání a vynechávání, podíváme se na hledání předchůdce a následníka v jednoduše zřetězeném seznamu.

Určení následníka (nebo zjištění, že neexistuje, jedná-li se o prvek na konci seznamu) je triviální, odpověď dává ukazatel na následující prvek. Proto se hledáním následníka zabývat nebudeme.

Zkusíme si tedy hledání předchůdce. Klepnutím pravým knoflíkem myši zvolte některý obdélník. Je také možno o náhodnou volbu prvku požádat Algovizi knoflíkem **Zvol prvek.** Zvolený prvek zmodrá.

Nyní zkuste nalézt předchůdce zvoleného modrého prvku. Jistě vám je jasné, že nezbývá než procházet seznamem od počátku tak dlouho, dokud nenarazíte na prvek, který na modrý prvek ukazuje. Vyhledávání lze provádět ručně, krokováním nebo animací. V prvním případě se, podobně jako v předchozí scéně, vycházejíce ze záhlaví, myší odkrývají ukazatele na další prvky seznamu, dokud se neobjeví ukazatel, který ukazuje na modrý prvek, jehož předchůdce hledáme. Předchůdce je pochopitelně ten prvek, ze kterého tento ukazatel vychází.

Pokud byste nechtěli postupovat systematicky a náhodně by se vám podařilo nalézt obdélník, jehož ukazatel ukazuje na zvolený modrý prvek seznamu, nejásejte. Na modrý prvek mohou ukazovat i některé matoucí obdélníky, které jsou i v této scéně přítomny, ale do našeho seznamu nepatří a tedy *nejsou předchůdcem* zvoleného modrého prvku v našem seznamu.

(Aby prvek, který náhodně zvolíte, patřil do našeho seznamu, je scéna naprogramována tak, že teprve po provedení volby Algovize provede rozdělení obdélníků na prvky seznamu a matoucí objekty tak, aby zvolený modrý prvek byl prvního typu, prvky seznamu nějakým způsobem propojí a ukazatele matoucích prvků nastaví tak, aby hodně z nich ukazovalo na modrý prvek. Hra je ale poctivá: takto situace skutečně *mohla* vypadat a od vaši první akce už Algovize ukazatele nemění a tedy nemá možnost situaci dodatečně ovlivňovat.)

Scéna: *Přidávání prvku zřetězeného seznamu*

U kompaktního seznamu bylo přidávání (s výjimkou přidávání na konec) obvykle zdlouhavou operací. Hlavní výhodou zřetězeného seznamu je, že alespoň některé varianty přidávání jsou velmi rychlé a vyžadují jen přepojení několika ukazatelů bez ohledu na to, jak je seznam dlouhý a v kterém jeho místě přidávání probíhá.

Základní rychlou variantou je u zřetězeného seznamu přidávání *za* zvolený prvek seznamu. Zvolte si klepnutím pravým knoflíkem myši prvek seznamu, za který se bude přidávat (tato scéna nevyužívá matoucích objektů). Prvek zmodrá a až do konce operace bude také zobrazen jeho ukazatel (i klíč, což ale není podstatné).

Nyní za zvolený prvek přidáme nový prvek s klíčem uvedeným na ovladači v pole vedle **Klíč.** Krokování operace se provede pomocí knoflíku **Krok.**

V prvním kroku se vytvoří nový objekt třídy, která popisuje prvky seznamu; místo, kde objekt vznikne, není důležité. Nový objekt je na obrazovce znázorněn červeně, hodnota jeho ukazatele je zatím nenastavena.

V druhém kroku se ukazatel červeného prvku nastaví tak, aby ukazoval do stejného místa jako ukazatel zvoleného prvku. Je to proto, že prvek, který byl dříve za modrým prvkem, by po přidání měl být za novým, nyní červeným, prvkem. Červený prvek ale ještě nepatří do seznamu, protože není ukazován žádným prvkem seznamu.

V třetím, posledním, kroku se ukazatel zvoleného modrého prvku přesměruje tak, aby ukazoval na nový červený prvek. Tím je přidávání hotovo.

Přepnete-li volbu na ovladači z **Krokování** na **Animace** přidávání se ukáže jako kratičká animace bez nutnosti používat knoflík **Krok** (který se ani neobjeví).

Zkuste ale přepnout na **Ručně.** Nyní je třeba provést operace ručně. Vytvoření nového objektu se provede klepnutím myší do prázdného místa obrazovky. Přepojení ukazatele se provádí tak, že se kurzorem najede na špičku šipky, představující ukazatel a se stisknutým knoflíkem myši se táhne nad objekt, na který má ukazovat. Pokud by se myš uvolnila nad prázdným místem, ukazatel se vrátí do původní polohy. Nulový nebo neinicializovaný ukazatel nemá šipku a v takovém případě se kurzor nastaví nad čtvereček, ze kterého by měla šipka ukazatele vycházet a pak se táhne do cílové polohy.

Můžete se zvolit několik úrovní nápovědy. V nižší úrovni vás Algovize bude kontrolovat a na chybný krok upozorní a vrátí jej zpět. V nejvyšší pak vám bude přímo říkat, co je třeba udělat.

Dejte si pozor na častou chybu: pokud bychom napřed přesměrovali ukazatel modrého objektu na červený, ztratíme nenávratně informaci o prvcích seznamu, které následovaly za zvoleným modrým objektem, což může vést k ztroskotání celého výpočtu, ve kterém je seznam používán.

Přidávání na začátek seznamu je svým způsobem speciální případ přidávání za zvolený prvek. Na záhlaví je možno dívat se jako na prvek seznamu,

který neobsahuje klíč. Přidávání na začátek je pak vlastně přidávání za záhlaví, které se provede výše uvedeným způsobem přepojením dvou ukazatelů.

Přidání nového prvku na konec seznamu je přidávání za (dosavadní) poslední prvek seznamu. Pokud jeho lokalizaci známe, je to rychlá operace, ale obecně jej musíme nejdříve nalézt a to je obvykle zdlouhavé.

Přidání nového prvku *před* daný prvek se provede tak, že se přidává *za* jeho předchůdce (v případě prvního prvku seznamu je možno za jeho předchůdce považovat záhlaví). Pokud bychom předchůdce daného prvku znali, je operace v mžiku hotova, ale naneštěstí je určení předchůdce většinou zdlouhavé a tedy je obecně zdlouhavá i operace přidávání před daný prvek.

Scéna: *Vynechávání prvku zřetězeného seznamu*

Napřed si zkusíme trochu nepřirozenou variantu vynechávání: vynechání prvku, který leží bezprostředně *za* zvoleným prvkem (je tedy jeho následníkem). Zvolte klepnutím pravým knoflíkem myši libovolný prvek seznamu. Prvek zmodrá a ukáže se jeho ukazatel. Navíc zčervená prvek ležící za zvoleným modrým prvkem a i jeho ukazatel bude viditelný. Červený prvek je ten, který má být vynechán. Pokud by se ukázalo, že zvolený modrý prvek byl poslední v seznamu, proveďte jinou volbu.

Operaci budeme nejprve krokovat. K jejímu provedení postačí jediný krok: ukazatel modrého prvku přepojíme tak, aby ukazoval na prvek, na který ukazuje ukazatel vynechávaného červeného prvku. Tím červený prvek vypadl ze seznamu a operace je provedena. Je vhodné upozornit na speciální případ: je-li ukazatel vynechávaného červeného prvku nulový (tedy prvek je na konci seznamu - ukazatel je znázorněn jako malé kolečko), pak se také vynuluje ukazatel modrého předchůdce.

Jelikož je operace tak krátká, Algovize nenabízí volbu animovaného provedení, ale zkuste si operaci provést ručně s přepojením ukazatele způsobem popsaným v předchozí scéně.

Pokud vynecháváme přímo zvolený prvek a ne až prvek za ním, může být (a většinou je) operace vynechávání zdlouhavá: nejprve se způsobem popsaným v předcházejících scénách nalezne předchůdce vynechávaného prvku (v případě prvního prvku seznamu je za něho považováno záhlaví) a pak se způsobem popsaným v této scéně zvolený prvek vynechá jako následník jeho již určeného předchůdce. Samotné vynechání je tedy jednoduché, ale určení předchůdce je v jednoduše zřetězeném seznamu obvykle časově náročné.

Scéna: *Rozdělení a spojení zřetězených seznamů*

V této scéně budeme pracovat se dvěma seznamy, proto máme k dispozici dvě záhlaví.

Rozdělení zřetězeného seznamu na dva je velmi rychlá operace: zvolte klepnutím libovolný prvek zobrazeného seznamu. Zvolený prvek zmodrá a

jeho následník zfialoví. Algovize seznam rozdělí na dva: první z nich je počáteční úsek původního seznamu, který končí zvoleným modrým prvkem a druhý seznam začíná fialovým následníkem zvoleného prvku a pokračuje až do konce původního seznamu.

Rozdělení se provede jednoduše: záhlaví druhého seznamu, které je tvořeno nulovým ukazatelem, protože druhý seznam je prázdný, se přepojí tak, aby ukazovalo na následníka zvoleného prvku, tedy na fialový první prvek odděleného seznamu. Pak se ukazatel na následníka ve zvoleném modrém prvku (který se má stát posledním prvkem prvního seznamu) vynuluje. Rozdělení seznamu je možno provádět v krokovacím, animovaném i ručním módu.

Rozdělené seznamy je možno upravovat, například přidávat prvky nebo je vynechávat, pokud se patřičně zvolí operace na ovladači. Nakonec ale zkusíme popřípadě pozměněné seznamy zase spojit. Tato operace by byla velmi rychlá, kdybychom znali poslední prvek seznamu, ke kterému připojujeme druhý seznam: ukazatel na následníka v posledním prvku prvního seznamu, který je v tomto okamžiku nulový, se přepojí tak, aby ukazoval na první prvek druhého seznamu (tedy prvek ukazovaný ze záhlaví druhého seznamu); potom se ukazatel v záhlaví druhého seznamu vynuluje. Jelikož ale u jednoduše zřetězeného seznamu obvykle ukazatel na konec není k dispozici (kvůli obtížím, které by nastaly při vynechání posledního prvku seznamu), musíme si konec nejprve nalézt, což je časově obecně velmi náročné. Spojení lze opět provádět ve všech třech módech.

Scéna: *Zřetězený zásobník*

Zásobník (neboli LIFO), tedy seznam s omezením přípustných operací, který umožňuje pouze přidávání a vynechávání na konci, se dá velmi výhodně implementovat zřetězeným seznamem. Jelikož ale již víme, že přidávání a vynechávání na konci je obecně obtížné, ale přidávání a vynechávání na začátku jednoduché, implementuje zásobník "pozpátku": ukazatel prvku nebude ukazovat na následníka, ale na předchůdce a proto ze záhlaví přímo vidíme konec seznamu. To co dříve bylo operací s prvním prvkem seznamu, je zde operací s posledním prvkem zásobníku. dalších komentářů jistě není třeba, zkuste si přidávání a vynechávání provádět (klíč přidávaného prvku je z pole **Klíč** na ovladači, který Algovize nastavuje náhodně, ale je ho možné před přidáním přepsat).

Zásobník se tedy i zřetězeným seznamem implementuje velmi jednoduše a efektivně a odpadá zde i obtíž s počáteční alokaci správně velkého pole, která nastává u kompaktního zásobníku.

Scéna: *Zřetězená fronta*

Fronta (neboli FIFO) přidává na konci a vynechává na začátku. Víme již, že vynechávání na začátku je jednoduché a rychlé. Přidávání na konci

je jednoduché a rychlé, pokud známe polohu posledního prvku seznamu. Implementujeme-li frontu, pak si explicitně polohu posledního prvku pamatujeme; kromě záhlaví (ukazatelen na první prvek seznamu) tedy v popisu fronty je i ukazatel na poslední prvek fronty, který bude umístěn pod záhlavím a bude šedivý. Pok bude přidávání na konec také rychlé (nezapoměňte přitom aktualizovat ukazatele na poslední prvek).

Můžete se zeptat, proč jsme ukazatel na poslení prvek nepoužívali u obecného seznamu, ale zavedeme jej u FIFO fronty. Je to proto, že potíž nastane, pokud se poslední prvek seznamu vynechá. Pak je potíž, kam ukazatel směrovat. Nový poslední prvek se pak musí určit zdlouhavým procházením seznamu od začátku. U FIFO fronty se ale poslední prvek nevynechává (pokud není zároveň prvním, tedy pokud ve frontě není jediný prvek), takže zmíněná obtíž nikdy nenastane.

I FIFO fronta se tedy zřetězeným seznamem implementuje velmi jednoduše a efektivně a i zde odpadá obtíž s počáteční alokaci pole.

Scéna: *Správa paměti*

Při vynechání prvku ze seznamu, tak jak jsme si to vyzkoušeli v minulé scéně, ztratíme odkaz na tu část paměti, ve které se vynechaný prvek nacházel. Počítačový systém, respektive s ním spojený alokátor paměti, tuto oblast neeviduje, protože ji předal uživatelské aplikaci. Aplikace ale přišla o odkaz tím, že ukazatel modrého prvku, který jediný na červený vynechávaný objekt ukazoval, byl přesměrován na jeho následníka (nebo vynulován, pokud tento následník neexistoval).

Budeme-li přidávání a vynechávání delší dobu opakovat v podobě z minulé scény, kdy se nestaráme o zpracování nebo recyklaci odpadu, můžeme brzo dospět do situace, že alokátor už odmítne vytvořit nový objekt, protože již vyčerpal všechnu volnou paměť, která byla aplikaci přidělena, ale aplikace s přidělenou pamětí špatně zacházela a všechnu ji zahodila do odpadu.

Tato scéna přináší ilustraci takové situace: přidávejte do seznamu nové prvky a již zařazené zase vynechávejte tak, že si nejprve na ovladači zatrhnete, kterou operaci chcete provést a pak provedete následující:

- Přidávání se provádí napůl ručně a napůl automaticky: napřed klepněte do volného místa a Algovize vám tam vytvoří nový objekt s náhodně zvoleným klíčem, pak klepněte na prvek seznamu, za který se bude přidávat (může to být i záhlaví) a Algovize nový objekt s animací přidá do seznamu za zvolený prvek.

- Vynechávání se spustí klepnutím na některý prvek seznamu. Algovize animovaným způsobem vynechá buď zvolený prvek nebo prvek za ním (pokud existuje) v závislosti na nastavení na ovladači (volba mezi **Zvolený** a **Za ním).** První možnost je to, co obvykle chceme, druhá je to, co vždy dovedeme provést bleskově.

Na rozdíl od minulé scény oblast, která byla obsazena vynechaným prvkem, takřka splyne s pozadím. Splynutí s pozadí vede k tomu, že při povrchním pohledu (například když se já dívám na obrazovku bez brýlí) vidíme jen ty segmenty paměti, ke kterým má aplikace přístup. V našem případě to konkrétně znamená, že vycházejíce ze záhlaví a postupujíce podél ukazatelů navštěvovaných objektů, se do dané oblasti můžeme dostat. Jelikož jsme při vynechání bezstarostně vyhodili referenci na vynechaný prvek, s oblastmi splývajícími s pozadím už nemůžeme pracovat, například do nich nelze klepnout, chcete-li vytvořit nový prvek seznamu. Nový prvek se nevytvoří ani pokud sice klepnete do volného místa, které je ale natolik sevřeno obsazenými oblastni, že se tak nevejde obdélník pro nový objekt. V obou případech by vás Algovize upozornila, proč nebyl nový objekt vytvořen.

Neúplnost splynutí s pozadím ale umožní při pozorném prohlížení obrazovky zjistit, jak situaci vidí alokátor paměti. O světlejších obdélníkových oblastech má poznamenáno, že je aplikaci přidělil. Pochopitelně neví, jak s nimi aplikace nakládá a tedy neví, že aplikace referenci na ně vyhodila, takže když je požádán o přidělení *nového* objektu pokouší se ho umístit do tmavší části volného prostoru, neboť pouze tu má stále označenu jako oblast, kterou má aplikace právo využívat, ale zatím ji nevyužívá.

Scéna začíná seznamem s pěti prvky. Doporučuji pak přidávat nové prvky a vynechávat již vložené tak, aby seznam měl stále přibližně stejnou velikost. Seznam pak stále zaujímá jen velmi malou část plochy obrazovky, která představuje oblast paměti, která vám (neboli uvažované aplikaci) byla systémem vyhrazena. Obdélníky pro konstrukci objektů jsou ale v této scéně voleny větší než ve scénách minulých a tak se velmi brzo stane, že světlejší část pozadí, představující oblast odpadu z vynechávání, o které si ale alokátor myslí, že ji aplikace smysluplně využívá, zabere tolik místa, že se vám do volné tmavé části pozadí již nepodaří nikde umístit nový objekt.

Celý postup lze také animovat; pak Algovize přidává a vynechává prvky seznamu sama a velmi brzo otráví volný prostor zbytky po vynechaných prvcích; pak vydá mezi programátory tolik "oblíbené" hlášení `"Out of memory"`.

Scéna: *Sběrač odpadu*

V této scéně uvidíme, jak je možno si počínat, abychom předešli nehospodárné správě paměti popsané v předchozí scéně.

Řešení problému správy paměti je jednoduché: zavedeme si ještě jeden seznam, kterému budeme říkat *odpad* (anglicky *garbage*). Jestliže ze seznamu vynecháme některý prvek, pak jej (dříve než zapomeneme referenci na jeho umístění) zařadíme do odpadu. V odpadním seznamu nezáleží na pořadí a proto je v případě jednoduše řetězených seznamů vhodné jej přidávat na začátek, protože to je nejjednodušší a nejrychlejší. Záhlaví pro odpadní seznam je na obrazovce umístěno hned pod záhlavím pracovního seznamu a je hnědé.

Stejně tak se na (světle) hnědou přebarví objekt v okamžiku jeho přepojení z pracovního seznamu do odpadu.

Pokud pak budeme chtít přidat do seznamu nový prvek, nepoběžíme hned žádat alokátor paměti o zbrusu nový objekt, ale nejprve se podíváme do odpadu. Pokud je odpadní seznam neprázdný (neboli v jeho záhlaví není nulový ukazatel) pak z odpadu vyjmeme *libovolný* prvek, do jeho datové oblasti zařadíme nově přidávaný klíč a případně další údaje a pak jej výše popsaným způsobem vložíme do pracovního seznamu. Z důvodu rychlosti a jednoduchosti výpočtu je výhodné pro recyklaci vyjmout první prvek odpadního seznamu, tedy prvek přímo ukazovaný ze záhlaví. Odpad tedy výhodně pracuje jako zásobník.

Pouze v případě, že seznam obsahující odpad z vynechávání je prázdný (což znamená, že veškerá paměť, která aplikaci byla alokátorem paměti přidělena, je užitečným způsobem využívána), požádáme alokátor o přidělení nového objektu.

Poznamenávám, že některé systémy nebo programovací jazyky (jako je třeba Java) seznam odpadu vedou za vás, aniž by se na cokoliv ptaly. Pro každý objekt, který byl aplikací vytvořen, počítají kolik ukazatelů v aplikaci na něj ukazuje, a pokud tento počet klesne na nulu, objekt zařadí do odpadu (přičemž uživatel mnohdy ani netuší, že byl takový odpadní seznam vytvořen). Alokace nových objektů pak využívá recyklace z odpadu a pouze když v odpadním seznamu nic není, obrací se na systémový alokátor. Tento komfort ovšem něco stojí - prosté přepojení ukazatele se stává poměrně komplexní akcí, která je provedena i když mnohdy vůbec není potřeba. Proto zkušený programátor vytvářející důležitý program pro hromadné použití mnohdy zvolí primitivnější a výkonnější programovací jazyk, ve kterém správu paměti související s používáním zřetězených seznamů vytvoří na míru aplikaci.

Poznamenávám také, že pokud používáme v programu více zřetězených seznamů, není třeba odpadní seznam vytvářet pro každý zvlášť, ale jeden pro všechny seznamy dohromady. Bez problémů se toto řešení použije pokud prvky všech seznamů jsou objekty stejného typu, ale dá se s jistými omezeními použít i pokud tato podmínka splněna není.

Používání odpadního seznamu se také neomezuje na jednoduše zřetězené seznamy, ale může být použito u mnoha dalších struktur, například u oboustranně zřetězených seznamů nebo binárních vyhledávacích stromů o kterých se v této knize ještě zmíníme. Objekty ve složitějších strukturách mívají více ukazatelů (ukazatel na předchůdce a následníka u oboustranně zřetězených seznamů, ukazatelé na syny a popřípadě i na otce u binárních stromů), ale odpad je možno i tam vést jako jednoduše zřetězený seznam, využívající jen jeden z dostupných ukazatelů, protože jediné operace s odpadem prováděné jsou přidání a vynechání na začátku (plus jednoduchý test neprázdnosti).

Scéna: *Operace s jednoduše zřetězenými seznamy*

V této scéně si můžete vyzkoušet všechny operace s jednoduše zřetězenými seznamy, které jsme probrali a to ve všech variantách a v ručním, krokovacím i automatickém módu. Je použit sběrač odpadu podobně jako v předchozí scéně, můžete jej ale nechat skrýt.

Scéna: *Dvousměrně zřetězené seznamy*

Viděli jsme, že některé operace z jednoduše zřetězenými seznamy jsou časové náročné z jednoduchého důvodu: časově náročné je v obecném případě nalezení předchůdce prvku jednoduše zřetězeného seznamu.

Předně se jedná o samotnou operaci určení předchůdce, která je užitečná například chceme-li procházet seznam pozpátku, od konce k jeho začátku; informace o předchůdci je ale mnohdy požadována sama o sobě.

Další operace, které určení předchůdce vyžadují, jsou vynechání zvoleného prvku seznamu a přidání nového prvku *před* zvolený prvek.

Určit rychle předchůdce bychom měli umět i pokud si chceme uchovávat explicitně informaci o posledním prvku seznamu. Není těžké zavést si kromě záhlaví seznamu i ukazatel na poslední prvek, ale horší je jej aktualizovat. Pokud se na konec seznamu přidá nový prvek, nic strašného se neděje, ukazatel na poslední prvek prostě přesměrujeme na nově přidaný prvek. Pokud ale poslední prvek máme vynechat, informaci o předchůdci potřebujeme nejen k provedení operace, ale i k tomu, abychom věděli, kam přepojit ukazatel na konec seznamu: dosavadní hodnota ukazatele se stala neaktuální a musíme jej přesměrovat na předchůdce původního konce.

Jestliže se operace vyžadující informaci o předchůdci provádějí častěji, je výhodné v každém prvku seznamu kromě ukazatele na následníka v seznamu používat i ukazatel na jeho předchůdce. Každý prvek seznamu tedy bude objekt třídy, která zahrnuje alespoň tři složky: ukazatel na následníka, ukazatel na předchůdce a klíč nebo obecněji datová složka objektu.

Tato scéna je přímou analogií předchozích scén a umožňuje provádět všechny výše popsané operace se seznamy ve všech jejich variantách a módech. Obdélníky ale jsou rozděleny na tři čtverečky odpovídající třem složkám objektu, vyjmenovaným v předchozím odstavci.

Záhlaví pracovního seznamu obsahuje dvě složky: ukazatel na první prvek a ukazatel na poslední prvek. Je též použit sběrač odpadu který, jak už bylo naznačeno ve scéně o sběrači odpadu, je implementován jako jednoduše zřetězený seznam, využívající pouze ukazatel na následníka (levý čtverec v objektu), zatímco ukazatel na předchůdce (střední čtverec) je nevyužíván a proto příslušná šipka není zobrazována.

Předpokládám, že operace s dvousměrně zřetězeným seznamem, které jsou co do manipulace s ukazateli na následníka identické s operacemi v jednoduše

zřetězeném seznamu, jsou natolik přirozené a zřejmé, že je není nutno komentovat. Dostačující je měnit ukazatel na předchůdce tak, aby předchůdce následníka i následník předchůdce libovolného prvku v seznamu (pokud existují) byli rovni prvku v.

Co se týče rychlosti provádění operací, musíme sice pro každý objekt, se kterým manipulujeme, pracovat se dvěma ukazateli, ale na druhou stranu v mnoha případech odpadá procházení seznamem nebo jeho podstatnou částí. Všechny základní varianty přidávání (na začátek, na konec, před a za zvolený prvek) a vynechávání (vynechání prvku na začátku nebo na konci, vynechání zvoleného prvku nebo prvku, který se nachází před nebo za ním) se provedou v konstantním čase, přepojením několika ukazatelů, bez ohledu na to, jak dlouhý seznam je. Stejně tak je okamžité určení prvního a posledního prvku seznamu a předchůdce a následníka zvoleného prvku seznamu. Rozdělení seznamu a spojení dvou seznamů v jeden jsou také operace, které lze provést v konstantním čase; u jednoduše zřetězených seznamů jsme viděli, že jedinou potíží je, že pro spojování je třeba znát konec seznamu, ke kterému se připojuje, což je ale zde přímo dostupná informace. Rychlá je zde i operace rozdělení, pokud je dán vrchol, který má být první v odpojovaném seznamu, protože jeho explicitně zadaný předchůdce bude poslední v prvním z výsledných seznamů.

Na druhé straně vyhledávání prvku s daným klíčem, hledání minima a maxima jsou operace, které ze samotné povahy problému vyžadují prohledání podstatné části seznamu a v nepříznivém případě celého seznamu a ani dvousměrně zřetězené seznamy neumožňují provádět tyto operace rychleji.

Kapitola 2

Binární stromy

V této kapitole se budeme zabývat datovými strukturami, které se používají pro uložení množin čísel, se kterými můžeme provádět následující operace:

- **inicializace** - vytvoří strukturu, popisující prázdnou množinu. Inicializaci také můžeme volat když chceme množinu vyprázdnit, t.j. najednou z ní vynechat vše, co se v ní nachází,
- **nalezení prvku** - je dáno číslo a má se zjistit, zda se v množině nachází a v kladném případě se také určí kde je uloženo (viz dále),
- **vložení prvku** - do množiny se vloží zadané číslo (pokud ovšem v množině již je, neprovede se nic),
- **vynechání prvku** - z množiny se vynechá číslo, zadané místem kde je uloženo (viz dále) - tím je zaručeno, že se v množině opravdu nachází; pokud nevíme, zda číslo, které chceme vynechat, v množině je a nebo kde je uloženo, musíme provést operaci nalezení prvku,
- **minimum**, **maximum** - pokud je množina neprázdná, naleznou nejmenší, resp. největší číslo, které v množině je.

Datová struktura implementující všechny tyto operace se obvykle nazývá *slovník*.

Jednoduchou možností, jak tyto operace provádět, je srovnat prvky do posloupnosti a posloupnost reprezentovat způsoby popsanými v předchozí kapitole. Podle toho, jak si prvky do posloupnosti seřadíme, existují dvě varianty tohoto postupu.

Jestliže připustíme jakékoli pořadí, pak je výhodné použít dvousměrně zřetězeného seznamu a přidávat například na začátek nebo na konec. Pak je velmi rychlé jak přidávání nového prvku (předpokládajíce, že v množině ještě

není) i vynechávání prvku množiny (pokud víme, kde je v seznamu uložen). Na druhé straně vyhledávání prvku v množině vyžaduje obecně projít celý seznam nebo jeho významnou část a proto je u velkých množin velmi zdlouhavé.

Pokud prvky množiny srovnáme monotónně (například rostoucím způsobem), pak použití zřetězených seznamů nepřináší nic zásadně nového: vynechávání prvku uloženého na známém místě je okamžité, neúspěšné vyhledávání není nutné dělat až do konce seznamu, ale jen do okamžiku, kdy již narazíme na čísla větší než je hledané číslo, ale na druhé straně nelze přidávat na začátek nebo konec, ale do přesně vymezeného místa v seznamu a tudíž operaci přidávání není možno provést v konstantním čase.

Zajímavá je ale implementace monotónním kompaktním seznamem. Přidávání a vynechání je sice neefektivní - místo, kde k němu dochází, může obecně ležet kdekoli v seznamu a proto se nevyhneme rozsáhlým přesunům prvků seznamu, ale je to jediná z implementací slovníkových struktur, ve které se efektivně vyhledává. V minulé kapitole, ve scéně o vyhledávání v uspořádaném kompaktním seznamu, jsme totiž viděli, že se operace dá provést v čase, který je úměrný logaritmu počtu prvků seznamu.

Jak je tedy vidět, pro každou implementaci slovníku seznamem je alespoň jedna z potřebných operací je prováděna časově neefektivním způsobem - v čase, který je v nepříznivém nebo nejhorším případě úměrný počtu prvků seznamu.

Stromové datové struktury jsou svým způsobem kombinací výhod vyhledávání půlením v kompaktním uspořádaném seznamu a rychlého vkládání a vynechávání v oboustranně zřetězeném seznamu. V této kapitole budeme probírat binární vyhledávací stromy, které budeme označovat zkratkou BVS. Množinu o n prvcích lze uložit do BVS tak, že libovolná z množinových operací bude provedena v čase úměrném logaritmu počtu prvků množiny. Celá záležitost má ale jeden háček: přidání a vynechání prvků se sice provedou rychle, ale ovlivní se tím tvar stromu způsobem, který závisí pouze na vztahu prvku, se kterým se pracuje, k ostatním prvkům množiny. Může se stát (a také se mnohdy stává), že se tvar stromu mění nepříznivým způsobem a v nejhorším případě strom zdegeneruje tak, že většina operací vyžaduje projít většinu (nebo dokonce všechny) vrcholy stromu a tím se provádění operací stane zdlouhavým a zhruba srovnatelným se seznamy.

Je sice možné ukázat, že pokud se klíče přidávají a vynechávají dostatečně náhodně (nebudu zde podrobněji rozebírat, co to znamená), degenerace tvaru stromu je obvykle velmi mírná a doba nutná k provádění operací není o mnoho delší než v optimálně zkonstruovaném stromu, ale často je potřeba mít *jistotu*, že strom příliš nezdegeneruje.

Zavádějí se proto varianty BVS, které pracují tak, že po přidání nebo vynechání prvku (nebo během něj) se kontroluje tvar stromu, a pokud je třeba, upraví se tak, aby nemohlo dojít k neefektivnímu provádění operací.

Dvě takové varianty, které se nazývají AVL-stromy a červeno-černé stromy, zde budou také probrány.

2.1 Binární vyhledávací stromy

Scéna: *Úvod*

Úvodní scéna je jen grafická ilustrace, ukazující binární strom o 500 vrcholech. Levý (pravý) syn libovolného vrcholu v a jeho potomci jsou zobrazeni vlevo (vpravo) od vrcholu v. Zkuste si ale přepnout mezi volbami **Náhled: Kořenový** a **Náhled: Lineární.** Při prvním se kořen umístí (horizontálně) do poloviny obrazovky, jeho synové do 1. a 3. čtvrtiny, jejich synové do 1., 3., 5. a 7. osminy obrazovky atd. V druhém případě jsou zase zleva doprava pravidelné rozestupy mezi x-ovými souřadnicemi vrcholů. Oba způsoby mají své výhody a nevýhody a oba budou používány, v mnoha případech bude mít uživatel volbu podobně jako v této scéně.

Scéna: *Binární strom*

Byl vytvořen binární vyhledávací strom (BVS) o 25 vrcholech. Podívejte na strom na obrazovce. Nejvyšší vrchol nazýváme *kořen* (strom tedy roste dolů, opačně než v přírodě). Každý vrchol může mít až dva *syny*, jeden z nich je *levý* a druhý je *pravý*; v případě, že vrchol má pouze jednoho syna, je určeno, zda je levý nebo pravý. Vrchol, který nemá žádného syna, se nazývá *list*. Je-li vrchol v synem vrcholu u, pak řekneme, že u je *otcem* vrcholu v. Každý vrchol stromu s výjimkou kořene má právě jednoho otce.

Jestliže existuje posloupnost vrcholů $v_0, \ldots, v_k$ taková, že pro $i = 1, \ldots, k$ je v_i synem vrcholu v_{i-1}, pak řekneme, že v_k je *potomkem* v_0 a naopak v_0 je *předkem* v_k. Připouštíme zde $k = 0$, neboli vrchol je také sám sobě potomkem i předkem.

Podstrom určený vrcholem u je množina obsahující všechny potomky vrcholu u. (Podstrom určený vrcholem tedy tento vrchol zahrnuje.) *Levý/pravý podstrom* vrcholu u je podstrom určený levým/pravým synem vrcholu u nebo prázdná množina, pokud tento syn neexistuje.

Klepněte myší na libovolný vrchol stromu; vybraný vrchol zrůžoví, ostatní vrcholy podstromu určeného vybraným vrcholem zůstanou tmavé, ale ostatní vrcholy vyblednou.

I v této scéně je možno přepínat mezi dvěma způsoby kreslení stromu - kořenovým a lineárním. Je možné si nechat vytvořit nový strom, který bude mít tolik vrcholů, kolik je číslo v poli u označení **Kolik vrcholů.**

Scéna: *Binární vyhledávací strom*

Množinu M obsahující n čísel uložíme pomocí BVS tak, že vytvoříme *libovolný* binární strom s n vrcholy a pak do každého vrcholu uložíme jeden prvek množiny M, zvaný *klíč*, a to tak, aby platilo pravidlo, uvedené v následující scéně. V této scéně je pouze ukázáno, jak klíče vrcholů znázorňujeme graficky: pod každým vrcholem je nakreslen sloupec, který svou výškou odpovídá velikosti klíče vrcholu. Klepněte na libovolný vrchol; zrůžoví vybraný vrchol a zároveň sloupec znázorňující velikost jeho klíče. V poli hodnot se také objeví modrá vodorovná čára, umožňující porovnávat snadno zvolený klíč s klíči ostatních vrcholů.

Do vrcholu se často ukládají někdy velice komplexní data, která jsou jednoznačně označena stanoveným identifikačním číslem, které pak slouží jako klíč vrcholu.

Velikost znázorněného BVS i jeho způsob nakreslení lze měnit stejně jako v předchozí scéně. Navíc je možno zvolit, zda bude zobrazováno okénko se sloupci znázorňujícími velikost klíčů ve vrcholech.

Scéna: *Binární vyhledávací strom - podmínka*

Klíče, popsané v předchozí scéně, musí být do vrcholů binárního stromu uloženy tak, aby platilo následující pravidlo, které zaručuje, že se ve stromu snadno nalezne hledaný klíč:

Pro libovolný vrchol v platí, že

- číslo uložené v libovolném vrcholu w, který patří do *levého* podstromu vrcholu v, je *menší* než číslo uložené ve v a
- číslo uložené v libovolném vrcholu w, který patří do *pravého* podstromu vrcholu v, je *větší* než číslo uložené ve v.

Zkontrolujte si platnost podmínky pro uvedený strom: Při klepnutí na libovolný vrchol tento vrchol zrůžoví, vrcholy jeho levého podstromu zmodrají a v pravém podstromu zezelenají. Ostatní vrcholy vyblednou. Podmínka stanoví, že klíče modrých vrcholů musí být menší než klíč růžového zvoleného vrcholu, klíče zelených vrcholů musí být větší než klíč růžového vrcholu. Při našem způsobu kreslení stromu podmínka znamená, že probíráme-li vrcholy zleva doprava, jejich klíče rostou, jak je vidět na sloupcích pod vrcholy.

Někdy se připouští, aby dva různé vrcholy měly stejný klíč a v podmínce se místo "menší/větší" říká "menší nebo rovno/větší nebo rovno", my to však pro jednoduchost výkladu připouštět nebudeme.

Scéna: *Vyhledávání v BVS*

Nyní se podíváme, jak zjistit, zda dané číslo k je klíčem některého z vrcholů stromu a v kladném případě tento vrchol určit.

Hledání začíná v kořeni stromu. Porovnáme hledané číslo s klíčem kořene. Jsou-li si rovny, hledaný vrchol byl nalezen. Tento případ je však velmi řídký. Pokud se čísla nerovnají, pak pravidlo pro rozmisťování klíčů v BVS říká, že je-li hledané číslo menší než klíč vrcholu, musíme přejít do levého syna kořene a hledat v jeho podstromu, je-li větší, musíme hledat v podstromu určeném pravým synem.

Hledání pak provádíme opakováním operace, která byla provedena v kořeni, tedy je-li klíč aktuálního vrcholu stejný jako hledané číslo, byl hledaný vrchol nalezen, jinak přejdeme do levého nebo pravého syna, pokud je hledané číslo menší, resp. větší než klíč vrcholu.

Může se ovšem stát, že bychom chtěli jít do levého (nebo pravého) syna a ten ve stromě není. To znamená, že se hledané číslo v množině popsané stromem nenachází, protože jinde než v chybějícím synu by být nemohlo.

Zkuste si nyní hledání čísla ve stromu jako klíče některého vrcholu: hledané číslo se zadá některým z následujícího způsobů:

- buď přímo numericky do pole vpravo od návěští **Klíč** na ovladači
- nebo klepnutím na některý vrchol - tím se jako číslo zvolí klíč poklepaného vrcholu
- nebo klepnutím do pole sloupečků pod stromem - hodnota je dána výškou bodu, zadaného klepnutím
- nebo klepnutím na **Dotaz** - počítač vybere dotazované číslo náhodně.

Ve všech čtyřech případech se zadaná hodnota zobrazí numericky na ovladači a graficky vodorovnou čárou v poli sloupců ve výšce úměrné zvolené hodnotě.

Průběh výpočtu lze sledovat krokováním (knoflík **Krok)** nebo jako plynulou animaci (knoflík **Animuj).**

Při hledání se do kořene umístí zelený žeton, který se pohybuje stromem a ukazuje kořen podstromu, ve kterém by se mohl skrývat hledaný klíč. Vrcholy nepatřící do tohoto podstromu (které určitě hledaný klíč neobsahují) vybledne. Je-li nalezen hledaný klíč, barva žetonu se změní na červeno-hnědou. V případě, že klíč nebyl nalezen, se žeton zbarví fialově a stranou od něj se zobrazí otazník, a to v místě, kde by měl být, ale není, další vrchol při hledání klíče.

Scéna: *Hledání minima v BVS*

Hledání minima je zřejmé a velmi jednoduché; z kořene žeton postupuje doleva dokud to jde. V okamžiku kdy se dostane do vrcholu, kterému schází levý syn, byl nalezen vrchol s minimálním klíčem. Způsob krokování operace je standardní.

Scéna: *Hledání maxima v BVS*

Hledání maxima je zrcadlovým obrazem hledání minima, žeton postupuje stále doprava.

Scéna: *Přidávání klíče do BVS*

Úvodní část této operace je stejná jako vyhledávání. Pokud se zjistí, že přidávaný klíč už ve stromu je, nic dalšího se neprovede. Pokud vyhledávání zjistí, že klíč ve stromu není, identifikuje zároveň místo, kde by se měl nacházet chybějící syn některého vrcholu. Pak se chybějící syn vytvoří a přidá a uloží se do něj přidávaný klíč.

Zadejte klíč podobně jako u operace vyhledávání, ale tak, aby klíč v množině nebyl (tedy např. při zadání klepnutím na vrchol pak číslo uvedené na ovladači pozměňte). Krokování se provádí standardním způsobem.

Scéna: *Vynechávání klíče listu*

Vynecháváme-li z množiny, kterou popisuje strom, jedno číslo, pak potřebujeme vědět, ve kterém se nachází vrcholu. To je často splněno; mnohdy je vynechávané číslo popsáno jako (neznámý) klíč zadaného vrcholu. Pokud ale místo uložení čísla ve stromu neznáme, pak napřed zavoláme operaci vyhledávání a příslušný vrchol určíme (může se stát, že zjistíme, že dané číslo ve skutečnosti není klíčem žádného vrcholu; pak pochopitelně neprovedeme nic).

V této scéně se zabýváme nejjednodušším případem, kdy vynechávané číslo leží v listu, tedy vrcholu bez synů. V tomto případě prostě list odtrhneme i s klíčem a vyhodíme.

Pokud vrchol měl otce, v závislosti na tom, zda byl levým či pravým synem svého otce, otci vynulujeme příslušný ukazatel na syna. Pokud vrchol neměl otce (byl tedy kořenem, což znamená, že vynechávaný vrchol byl jediným vrcholem stromu), bude po vynechání strom prázdný, což obvykle znamená, že je nutno upravit referenci na strom (kterou bývá ukazatel na nyní neexistující kořen).

Kliknutím zvolte některý list (kliknutí je neaktivní u vnitřních vrcholů stromu) a pak operaci vynechávání buď proveďte přímo nebo si ji zkuste krokovat.

Scéna: *Vynechávání klíče z vrcholu s jedním synem*

Tato scéna je podobná jako předchozí, ale předpokládáme, že vrchol určený k vynechání má právě jednoho syna (buď levého nebo pravého). Tato operace je o něco složitější.

Je-li vynechávaný v vrchol kořenem (nulový otec), pak se vrchol v odtrhne a jeho (jediný) syn w se stane novým kořenem stromu.

V opačném případě se vrchol v také odtrhne, ale jeho jediný syn w je jako syn adoptován otcem vrcholu v (a stane se levým nebo pravým synem podle toho, zda v byl levým nebo pravým synem svého otce, ale bez ohledu na to, jakým byl w synem vynechaného vrcholu v).

Klikněte na vrchol s jediným synem (pro ostatní vrcholy je kliknutí neaktivní). V této scéně je zaručeno, že strom takový vrchol bude mít (vrcholy s jedním synem jsou na rozdíl od listů a vrcholů s oběma syny u náhodně vytvářených stromů málo časté). Pak si operaci vynechávání proveďte; doporučuji si animaci kroku přepojování ukazatele zpomalit.

Scéna: *Vynechávání klíče z vrcholu s oběma syny*

Nakonec nejsložitější je vynechání vrcholu v se dvěma syny. Takový vrchol nelze přímo vynechat, jeho dva syny by nebylo možno na jeho otce napojit. Proto postupujeme jinak: z vrcholu v napřed odstraníme klíč; bylo to číslo, které máme odstranit. Potom nalezneme vrchol w, jehož klíč je nejblíže nižší vynechanému klíči a klíč z w přesuneme do v a nakonec vynecháme vrchol w.

Volba nejblíže nižšího klíče je diktována tím, aby přesunem klíče z w do v se neporušila podmínka o umístění klíčů v BVS (bylo by také možno volit klíč nejblíže vyšší).

Snadno se nahlédne, že vrchol s nejblíže nižším klíčem se nalezne tak, že se z v přejde do jeho levého syna (mějte na paměti, že v má oba syny) a pak se opakovaně postupuje do pravého syna, dokud existuje. Je to vlastně hledání maxima v levém podstromu vrcholu v, kde se nacházejí klíče menší.

Ze způsobu hledání vrcholu s nejblíže nižším klíčem také plyne, že nalezený vrchol w nebude mít pravého syna (a možná ani levého), takže jej již umíme vynechat jednou z výše uvedených operací, popsaných v předchozích dvou scénách.

Scéna: *Vynechávání klíče z BVS*

Pro úplnost si ještě ukážeme vynechání klíče, který ale není zadán volbou vrcholu ve kterém leží, ale číselnou hodnotou; zadávání dotazu je obdobné jako při přidávání. Je to vlastně kombinace hledání klíče a vynechávání klíče daného polohou.

Scéna: *Operace s BVS*

V této scéně je možno si vyzkoušet všechny operace s binárním vyhledávacím stromem; stačí zvolit si na ovladači operaci a standardním způsobem ji provést nebo krokovat. Vrchol pro vynechávání je nutno zvolit klepnutím myší.

Scéna: *Hloubka náhodného BVS*

Libovolná z výše uvedených operací se provádí podél cesty, která vede z kořene do některého z vrcholů (například nalezený vrchol u vyhledávání, minima a maxima, přidaný vrchol u přidávání) nebo mezi některým vrcholem a jeho potomkem (otec a syn vynechávaného listu nebo vrcholu s jedním synem nebo cesta z vrcholu do maxima jeho levého podstromu při vynechávání vrcholu s dvěma syny). Na každé hladině přitom provedeme pouze několik málo operací, jejichž počet je omezen konstantou nezávislou na velikosti stromu. Proto je doba potřebná k provedení libovolné z výše popsaných operací i v nejhorším případě úměrná hloubce stromu, t.j. délce nejdelší cesty z kořene do nějakého listu.

Hloubka BVS může být velmi malá ve srovnání s počtem jeho vrcholů. V optimálním případě, kdy s výjimkou nejnižší vrstvy vrcholů má každý vrchol 2 syny, je v nejvyšší vrstvě (budeme ji označovat jako nultou) kořen, tedy 1 vrchol, v první vrstvě 2 vrcholy a v každé další vrstvě dvojnásobek vrcholů vzhledem k předchozí vrstvě, tedy v j-té vrstvě je 2^j vrcholů a má-li strom k vrstev, pak obsahuje $1+2+4+\cdots+2^k = 2^{k+1}-1$ vrcholů, tedy jeho hloubka k je rovna $\lceil \log_2(n+1) \rceil - 1 \approx \log_2 n$.

Jistě víte, že funkce logaritmus roste opravdu velmi pomalu; například $\log_2 1000000000$ je méně než 30 (a strom s miliardou vrcholů se nám už těžko vejde do počítače).

Jestliže prvky přidáváme do binárního vyhledávacího stromu náhodně (například u appletu náhodně se stejnou pravděpodobností z jistého velkého intervalu celých nezáporných čísel), pak hloubka stromu není o mnoho větší, než u optimálního stromu (dá se dokázat, že je nejčastěji okolo $1,44 \log_2 n$).

Knoflíkem **Přidej** přidávejte do původně prázdného stromu najednou větší množství náhodně a rovnoměrně generovaných klíčů. Počet pro jednu dávku je nastaven na 500, ale je možné ho na ovladači změnit. Je vidět, že strom je obvykle poměrně dobře vyvážený.

Hodnoty v pravém horním rohu znamenají: N celkový počet vrcholů, $maxH$ hloubku stromu (délka *nejdelší* cesty z kořene do listu) a $aveH$ průměrnou hloubku vrcholu (tedy aritmetický průměr délek cest z kořene do jednotlivých vrcholů - všech, ne jen listů). Je patrno, že i velmi velké stromy mají malou hloubku. Nakonec $\log_2 N$ znamená dvojkový logaritmus čísla N pro ilustraci, že poměr maximální i průměrné hloubky náhodně vytvářeného stromu k $\log_2 N$ zůstává přibližně stejný (a malý).

Je také možné změnit způsob kreslení stromu přepnutím mezi **Náhled: Lineární** a **Náhled: Kořenový.** Druhý z nich pěkně ukazuje, že některé větve stromu bývají zakrnělé, ale obvykle se nevyskytují větve příliš přerostlé.

Scéna: *Hloubka náhodného BVS (pokračování)*

Tato scéna je prakticky stejná jako předchozí, vrcholy jsou přidávány po

100, ale klíče jsou generovány jiným způsobem: klíč je vybrán náhodně a rovnoměrně v rozmezí od 0 do B, kde B pozvolna roste. Způsob růstu není příliš důležitý, je volen tak, aby vytvářel zajímavě vypadající obrázky, ale podstatné je, že se *střední velikost* generovaných klíčů postupně *zvětšuje*.

Knoflíkem **Přidej** opětně přidávejte vrcholy a sledujte tvar stromu. Ze začátku ještě strom vypadá celkem vyváženě, pak ale začne pravá větev výrazně růst na úkor druhých, takže maximální i střední hloubka vrcholu, udané čísly $maxH$ a $aveH$ jsou výrazně větší než v předchozí scéně pro srovnatelný počet vrcholů N.

Scéna: *Vyhledávání v nevyváženém BVS*

V této scéně pouze zkusíme krok po kroku vyhledat náhodně zvolený klíč vrcholu ve stromu s 2000 vrcholy, který byl generován způsobem popsaným v předchozí scéně. Vyhledávání je obvykle tak zdlouhavé, že budete-li postupovat po jednotlivých krocích, patrně je ani nedokončíte.

Scéna: *Rotace*

V předchozích třech scénách jsme viděli, že tvar binárního vyhledávacího stromu může zdegenerovat tak, že operace s ním budou probíhat velmi pomalu. Proto se zavádí různé způsoby vyvažování, které udržují tvar binárního stromu v přijatelných mezích a proto i v nejhorším případě operace s nimi probíhají velmi rychle. Základní operací, kterou se vyvažují binární stromy (AVL-stromy a červeno-černé stromy, o kterých bude řeč později) je *rotace*.

Klepněte na některou hranu stromu na obrazovce. Zvolená hrana se "překlopí" neboli "rotuje" a tím se změní tvar stromu. Opětovným klepnutím na překlopenou hranu ji překlopíme do původní polohy - rotace je vratná operace.

Velmi doporučuji nastavit rychlost animace tak, aby bylo možno sledovat, co se při rotaci děje.

Důležité je, že při rotaci se vrcholy při našem způsobu zobrazování stromů posouvají pouze vertikálně, takže zůstává v platnosti podmínka na rozmístění klíčů v BVS, kterou je možno formulovat tak, že procházíme-li vrcholy zleva doprava (podle x-ové souřadnice), jejich klíče rostou.

Dobré pochopení rotace je podstatné pro pochopení funkce AVL-stromů i červeno-černých stromů, proto tuto a následující tři scény neopouštějte brzo, ale pohrajte si se stromem. Zkuste například dostat rotacemi vrchol z nejnižší hladiny nahoru, aby se stal kořenem, nebo naopak spustit kořen až dolů, aby se stal listem.

Scéna: *Rotační schéma*

Tento obrázek, který můžete spatřit ve všech učebnicích, které o BVS pojednávají, ukazuje rotaci schematicky. Klepejte na knoflík **Rotace.** Hrana

uprostřed se překlápí; z jejího horního konce vychází kromě rotující hrany ještě vazba na druhého syna horního vrcholu a jeho podstrom (naznačený jako trojúhelníková oblast) a z dolního konce vycházejí hrany ke dvěma synům a jejich podstromům. Povšimněte si, že při překlopení trojúhelník představující levý podstrom se pohybuje vertikálně opačným směrem než pravý podstrom a podstrom uprostřed zůstává ve stejné výšce. Tento nepohyblivý strom nám v některých případech zkomplikuje návrh vyvažovacích operací.

Scéna: *Rotace ve velkém stromu*

Scéna ukazuje totéž jako rotační schéma z předchozí scény, ale ne schematicky, nýbrž na příkladě velkého stromu. Klepněte na libovolnou hranu; pro názornost je lepší volit dlouhou hranu nahoře blízko kořene. Je dobře vidět v opačných směrech vertikálně se pohybující podstromy a nehybný podstrom mezi nimi.

2.2 AVL stromy

Scéna: *AVL strom*

Aby se předešlo vzniku nevyvážených BVS, ve kterých provádění operací trvá velmi dlouho, přidávání a vynechávání vrcholů se upravuje různými způsoby, které zabraňují vzniku výrazných nevyvážeností. Jednou z těchto úprav jsou AVL stromy (pojmenované podle autorů metody, Adelson-Velského a Landise).

AVL-strom je binární vyhledávací strom, který pro každý svůj vrchol v splňuje následující podmínku:
výška levého podstromu vrcholu v se od výšky jeho pravého podstromu liší nejvýše o 1.

Speciálně pokud jeden syn některého vrcholu chybí, druhý chybí také nebo je listem.

Vrcholy AVL-stromu budeme znázorňovat jako vahadélka. Jestliže pro jistý vrchol je jeho levý podstrom (t.zn. podstrom určený jeho levým synem) stejně hluboký jako jeho pravý podstrom, je vahadélko vodorovné. (Speciálně je vodorovné, pokud vrchol postrádá jak levého, tak i pravého syna.) V opačném případě se vahadélko nakloní na stranu hlubšího podstromu.

Ovladač je na počátku nastaven na přidávání vrcholů po jednom: knoflíkem **Další vrchol** přidávejte vrcholy a sledujte, že strom stále splňuje AVL pravidlo. Změnou čísla v poli **Kolik vrcholů** se nastaví počet v jednom kroku přidaných vrcholů. Co se při přidávání děje bude vysvětleno později.

Zkuste si strom vymazat knoflíkem **Zruš strom** a změnit způsob náhodné volby přidávaných čísel na **Posunující se distribuci,** která má devastující

účinky u prostého BVS. U AVL stromu ke vzniku nevyváženosti nedochází, vyváženost stromu je lepší, než byla u stejnoměrně náhodného rozložení vyváženost BVS.

Můžete též strom vymazat znovu a změnit způsob náhodné volby přidávaných čísel na **Monotónní distribuci.** Ta přidává čísla z rostoucí řady $0, 1, 2, \ldots$. U prostého BVS by to vedlo k degeneraci stromu na jedinou větev, hloubka stromu by byla největší možná (max. hloubka vrcholu $N - 1$, průměrná $(N - 1)/2$). Ani zde nedojde u AVL stromu ke vzniku podstatnější nevyváženosti. Je také dobře vidět, jak vrcholy se stále objevují na pravé straně, ale jakmile se pravá větev trochu prodlouží, vrcholy se "přelijí" nebo "překlopí" nalevo a strom se vyváží.

Scéna: *Rotace v AVL stromu*

Tato scéna je opakováním scény s rotací hran, ale strom je zobrazován jako AVL strom s vahadélkovými vrcholy.

Znázorněný strom ale nemusí a obvykle není AVL-strom, protože AVL-podmínka není v některých vrcholech splněna. Vrcholy, které ji porušují, jsou také nakloněny na stranu hlubšího podstromu, ale navíc ještě červeně obroubeny. Pokud takové vrcholy nejsou přítomny, vygenerujte si nový strom.

Zkuste si opět provádět rotace. Rotacemi lze dosáhnout takřka libovolných změn tvaru stromu. Zůstaňte u této scény nejméně tak dlouho, dokud strom nepřeformujete tak, aby to byl AVL-strom, tedy aby neměl žádný červeně obroubený vrchol. Uvidíte, že to poměrně snadné, ale nikoliv triviální.

Pokud červeně obroubené vrcholy odstraníte, zkuste naopak AVL-podmínku hodně pokazit. V nejhorším případě pouze dva vrcholy nejsou červeně obroubené - zkuste takový tvar vyrobit. Rotace je vratná, dá se jí tvar stromu zkazit, ale pak zase napravit. AVL algoritmus pochopitelně používá rotace uvážlivě, aby vyváženost zachovával, nikoli kazil.

Pak ještě znovu silně nevyvážené vrcholy "vyrotujte" pryč. Celý postup třeba několikrát zopakujte, dokud vám nebude zcela jasné, co to je AVL-strom a jak ho získat rotacemi.

Scéna: *Přidávání do AVL-stromu - případ 0*

Po seznámení se s rotacemi se již můžeme pustit do operací AVL stromu. Jelikož vyhledávání a určování minima a maxima nemění tvar stromu, provádějí se stejně jako v BVS. Operace, které ale tvar mění a obecně mohou porušit podmínku AVL, jsou přidávání a vynechávání klíče nebo vrcholu. V této scéně si probereme operaci přidávání do stromu, který Algovize vytvořila.

Do AVL stromu se nejprve přidá nový vrchol se zadaným klíčem úplně stejně jako u BVS stromu. Poté mohou nastat tři případy. V této scéně je probírán nejjednodušší z nich, kdy se po přidání nového vrcholu BVS algoritmem neporuší podmínka AVL a proto není třeba tvar stromu měnit.

I když ale po přidání vrcholu je strom stále AVL-stromem, operace ještě (na rozdíl od BVS stromů) nekončí. Povšimněte si, že po přidání vrcholu se u některých vrcholů na cestě od přidaného vrcholu zpět ke kořeni objeví rozdvojení. Pod černým vahadélkem vrcholu se objeví neúplně zakryté bílé vahadélko, skloněné pod jiným úhlem. Vysvětlím nyní, co to znamená.

Při zjišťování, který vrchol je popřípadě silně nevyvážený nebudeme vždy znovu určovat výšky podstromů jeho synů, protože by to bylo příliš zdlouhavé. Namísto toho bude každý vrchol zahrnovat proměnou, které má hodnoty "nakloněn vlevo", "vyvážený" a "nakloněn vpravo". Pokud se vyváženost vrcholu změní, musíme hodnotu proměnné opravit.

Přidáním listu do AVL-stromu se vyváženost řady vrcholů změní, ale chvíli potrvá, než příslušným způsobem opravíme hodnoty proměnných, udávajících u jednotlivých vrcholů jejich vyváženost. Tyto proměnné jsou tedy po jistou dobu neaktuální, což je vyjádřeno dvojitými vahadélky. Černá vahadélka v popředí vyjadřují skutečný stav a proto se jejich polohy okamžitě změní po přidání listu. Bílá vahadélka v pozadí označují hodnoty proměnných, uložené v počítači; jejich hodnoty zůstávají po jistou dobu nezměněny a proto u řady vrcholů nesprávné. Vzhůru stoupající zelený žeton je aktualizuje. Z tohoto důvodu nemůžeme přidávání okamžitě po přidání listu ukončit, ani kdyby AVL-podmínka zůstala zachována.

Jistě poznáte, proč někdy zelený žeton, označující aktivní vrchol, vystoupá až ke kořeni a jindy se poměrně brzo zastaví. Žeton se do vrcholu dostane z podstromu, jehož hloubka se přidáním zvětšila. Musí se proto opravit hodnota proměnné, udávající vyváženost vrcholu.

Byl-li vrchol vyvážený (tedy bílé vahadélko je vodorovné), nakloní se na stranu odkud žeton přišel (a tím se bílé vahadélko, udávající hodnotu balanční proměnné, skryje za černé vahadélko, udávající skutečný stav). V tomto případě se zjistí, že se hloubka podstromu zakořeněného ve vrcholu, ve kterém se nachází zelený žeton, zvýšila. Proto musí žeton postoupit výše a upravit údaj o vyváženosti i tam.

Byl-li vrchol nakloněn na opačnou stranu, než odkud přišel zelený žeton, stane se vyváženým (původně nakloněné bílé vahadélko se skryje za vodorovné černé vahadélko). V tomto případě se ale hloubka podstromu zakořeněného ve vrcholu, ve kterém se nachází zelený žeton, nezvýšila. Údaje o vyváženosti předchůdců vrcholu proto jsou v pořádku a operaci je možno ukončit.

Kdyby byl vrchol nakloněn na stranu, odkud přišel zelený žeton, stal by se silně nevyváženým. Strom a dotaz v této scéně byly vytvořeny tak, aby k této situaci nedošlo. Setkáme se s ní ale ve scénách následujících.

Scéna: *Přidávání do AVL-stromu - případ 1*

Případ 1 nebo 2 nastávají, pokud přidáním vrcholu podle BVS algoritmu vznikne alespoň jeden silně nevyvážený vrchol - tak budeme nazývat vrchol,

který porušuje AVL podmínku. Takových vrcholů může vzniknout víc, příklad v této scéně je volen tak, aby jich bylo hodně. Jistě ale postřehnete, že jsou vždy na cestě spojující kořen a přidaný vrchol.

V případě 1 i 2 zelený žeton po přidání nového vrcholu stoupá vzhůru, upravuje hodnoty proměnných popisujících vyváženost vrcholů (což je graficky znázorněno jako srovnávání bílých vahadélek s černými) a to tak dlouho, dokud nenarazí na vrchol, který již před přidáním byl nakloněn na stranu, ze které nyní do něho vstoupil žeton. Tento vrchol se po přidání stal silně nevyváženým a je nyní červeně obrouben. Označme tento vrchol, který je nejnižší ze silně nevyvážených vrcholů, jako v a označme jako w toho z jeho synů, ke kterému je vrchol v nakloněn.

Pro usnadnění identifikace vrchol v v okamžiku, kdy do něho vstoupí žeton, zfialoví a jeho syn w zmodrá.

Případ 1, probíraný v této scéně nastává, když buď vrchol w je *vyvážen* a nebo jsou vrcholy v a w nakloněny na *stejnou stranu*.

V případě 1 stačí provést rotaci hrany spojující v s w. Tím se spraví nejen vyvážení vrcholu v, ale i všech případných silně nevyvážených vrcholů nad ním.

Podmínka tohoto případu totiž říká, že nevyváženost vrcholu v způsobuje nízký podstrom druhého syna vrcholu v než je w a případně příliš vysoký podstrom syna vrcholu w, který je vzdálenější od v. Při rotaci se ale nízký podstrom prodlouží a příliš hluboko zasahující podstrom naopak stoupne a tím se rovnováha v původním podstromu vrcholu v (jehož novým kořenem je nyní w) obnoví. Navíc si všimněte, že se podstromu zakořeněnému ve v přidáním vrcholu zvýšila hloubka o 1 (což způsobilo vznik silně nevyvážených vrcholů nad v), ale rotací se jeho hloubka vrátila na původní velikost (po rotaci je ovšem jeho kořenem w) a tím se silně nevyvážené vrcholy nad v zase spravily.

Scéna: *Přidávání do AVL-stromu - případ 2*

Situace v této scéně i průběh operace jsou velmi podobné scéně předchozí; i zde označíme nejnižší vrchol, který se stal silně nevyváženým, jako v a w bude jeho syn, ke kterému je v nakloněn.

Případ 2, probíraný v této scéně, nastává když vrcholy v a w jsou nakloněny na *opačné strany*.

Označme ještě jako z toho syna vrcholu w, ke kterému je w nakloněn. Podobně jako v předchozí scéně vrchol v v okamžiku, kdy do něho vstoupí žeton, zfialoví a vrchol w zmodrá, navíc vrchol z zhnědne.

V tomto případě by samotná rotace hrany $v - w$ silnou nevyváženost vrcholu v neodstranila. Ta je totiž způsobena tím, že příliš hluboko zasahuje podstrom syna z vrcholu w, který se při rotaci nepohybuje a proto dosahuje stále stejně hluboko.

V tomto případě se proto provede nejprve rotace hrany $w - z$. Po rotaci je sice v stále silně nevyvážený, ale namísto případu 2 nastane nevyváženost z případu 1 z minulé scény a následná rotace hrany $v - w$ tedy ze stromu učiní opět AVL strom.

I zde dvojrotace obnoví vyváženost předchůdců vrcholu v a proto je po jejím provedení možno provádění operace ukončit.

Scéna: *Vynechávání vrcholu AVL-stromu*

Vynechávání vrcholu je jako obvykle nejkomplikovanější operací. Některé úkony se provádějí stejně jako u BVS. Vynechávání klíče daného hodnotou se převede na nalezení jeho polohy ve stromu a následné vynechání klíče daného polohou. Vynechání klíče z vrcholu se dvěma syny se převede na vynechání klíče z vrcholu bez pravého syna stejně jako se to provádělo u BVS.

Vynechávání vrcholu s nejvýše jedním synem se ale musí upravit. Ať je vynecháván list nebo vrchol s jedním synem, otci vynechávaného vrcholu se sníží výška podstromu jednoho z jeho synů.

Podobně jako u přidávání vrcholu, i zde se po odtržení vrcholu objeví rozdvojená vahadélka v řadě vrcholů na cestě z místa vynechání ke kořeni, která indikují, že proměnné popisující vyváženost vrcholů, znázorněná bílými vahadélky, neodpovídají aktuálnímu stavu, znázorněnému černými vahadélky. Proto zelený žeton, udávající aktivní vrchol, začne stoupat vzhůru počínaje otcem vynechaného vrcholu a proměnné vyvažuje, což se graficky projeví jako natočení bílého vahadélka do zákrytu s černým.

Může ovšem vzniknout i silně nevyvážený vrchol; na rozdíl od přidávání ale může vzniknout jen jeden a v žádném vrcholu nad ním již nejsou rozštěpená vahadélka (proměnné popisující vyváženost jsou v pořádku). Silná nevyváženost některého vrcholu v totiž vznikne tím, že se sníží hloubka stromu toho syna vrcholu v, od kterého byl jeho otec v odvrácen. Hloubka stromu, zakořeněného ve v, je ale dána hloubkou podstromu toho syna, ke kterému je v nakloněn a ta se nemění a proto se nezmění ani hloubka podstromu zakořeněného ve v a proto se nezmění ani hodnoty vyváženosti předchůdců vrcholu v.

Předpokládejme, že silně nevyvážený vrchol vznikl, označme jej v, syna vrcholu v, ke kterému je v nakloněn, označíme w. Pro usnadnění identifikace vrcholů Algovize nechá v okamžiku, kdy do v vstoupí zelený žeton, vrchol v zfialovět a w zmodrat. Nyní může nastat jeden z následujících případů:

Případ 1a, kdy vrchol w je vyvážený. Provedeme rotaci hrany $v - w$. Tou se silná nevyváženost vrcholu v odstraní. Navíc podstrom původně zakořeněný ve vrcholu v, jehož kořenem je po rotaci vrchol w, nezmění svoji výšku a proto není třeba upravovat proměnné popisující vyváženost předchůdců tohoto podstromu a celá operace končí.

Případ 1b nastává, když vrchol w je nakloněn na stejnou stranu jako vrchol

v. I zde provedeme rotaci hrany $v-w$. Na rozdíl od případu 1a, kde výška podstromu, původně zakořeněného ve vrcholu v, závisí na podstromech *obou* synů vrcholu w, tedy i toho, který se při rotaci nepohybuje, zde závisí jen na tom podstromu vrcholu w, který při rotaci stoupá vzhůru. Hloubka podstromu, zakořeněného původně ve v a po rotaci ve w, se proto *sníží*. To vede k tomu, že u řady předchůdců podstromu přestanou být aktuální hodnoty proměnných udávajících vyvážení. Navíc může dojít u některého z předchůdců podstromu ke vzniku silné nevyváženosti. V takovém případě ze stejných důvodů jako výše silně nevyvážený vrchol bude jediný. Operace proto musí pokračovat; žeton se pohybuje vzhůru, aktualizuje hodnoty vyváženosti vrcholů a pokud dorazí do silně nevyváženého vrcholu, provede se vyvážení podle jednoho z případů 1a, 1b nebo 2.

Případ 2 nastává, pokud vrchol w je nakloněn na opačnou stranu než vrchol v. V tomto případě syn z vrcholu w, ke kterému je w nakloněn, při vstupu žetonu do vrcholu v zhnědne. Podobně jako při přidávání, i zde se nejprve provede rotace hrany $w-z$, čímž se případ 2 převede na jeden z případů 1a a 1b a dále se postupuje jako výše, s tím, že i zde odstranění nevyváženosti vrcholu v může vést ke vzniku nových neaktuálností a/nebo silné nevyváženosti.

Při vynechávání klíče vrcholu tedy se může stát, že bude potřeba provést větší množství rotací nebo dvojrotací. Každý nový silně nevyvážený vrchol ale vznikne ve *vyšší* hladině na cestě od vynechaného vrcholu ke kořeni a proto je počet celkově provedených aktualizací a rotací dohromady nejvýše roven délce cesty od vynechaného vrcholu ke kořeni a tedy je shora omezen hloubkou stromu.

Opakovaně si zkuste vynechání klíče z vrcholu. Vrchol určený k vynechání zvolte kliknutím, můžete si také nechat vygenerovat nový strom s počtem vrcholů zadaným na ovladači. Při odstraňování silné nevyváženosti je na obrazovce uvedeno, který případ nastává. Nepostupujte dále, dokud neproberete všechny případy, které mohou nastat.

Scéna: *Operace s AVL*

Podobně jako tomu bylo u BVS, v této scéně je možno si vyzkoušet všechny výše popsané operace s AVL stromy.

2.3 Červeno-černé stromy

Scéna: *Červeno-černý strom*

Nyní bude ukázán jiný způsob, jak předejít vzniku příliš nevyváženého stromu, t.zv. červeno-černý strom. Tato scéna se obsluhuje přesně stejně jako v úvodní scéně AVL-stromů, ale je použit jiný typ stromu.

Jak je vidět, název stromu pochází z toho, že se jedná o binární vyhledávací strom speciálního tvaru, pro jehož popis je každý vrchol stromu obarven červenou nebo černou barvou.

U červeno-černého stromu musí platit následující podmínky:

ČČ1 každá cesta začínající v kořenu a končící ve vrcholu, kterému schází jeden nebo oba synové, musí obsahovat stejný počet černých vrcholů

ČČ2 syn červeného vrcholu nesmí být červený

ČČ3 kořen stromu je černý.

Vrcholu, kterému schází jeden nebo oba synové, budeme říkat *neúplný* vrchol.

Počet černých vrcholů na cestě z kořene do nějakého vrcholu v budeme nazývat *černé skóre* vrcholu v. ČČ1 tedy říká, že černá skóre neúplných vrcholů jsou stejná.

Povšimněte si, že ČČ1 implikuje, že černé skóre *neúplného* vrcholu v nemůže být menší než černé skóre *libovolného* jiného vrcholu w, protože cesta z kořene do w se dá protáhnout na cestu z kořene přes w do nějakého neúplného vrcholu z a cesty z kořene do v a do z musí mít stejně černých vrcholů. Takto upravenou podmínku ČČ1 budeme často používat a budeme ji označovat ČČ1'.

Podmínky implikují, že poměr délek nejkratší a nejdelší mezi cestami z kořene do neúplného vrcholu je nejvýše 2: černých vrcholů je na takových cestách stejně (ČČ1) a červených nemůže být více než černých (ČČ2 a ČČ3). Červeno-černý strom je proto poměrně dobře vyvážený.

Podmínka ČČ3 je nepodstatná, pokud by nebyla splněna, stačí kořen začernit; tím se ani ČČ1 ani ČČ2 neovlivní (kořen je jediný vrchol, který leží ve všech cestách uvažovaných v ČČ1 a proto jeho barvu lze libovolně měnit, aniž by tím ČČ1 byla ohrožena a začernění vrcholu nemůže ohrozit ČČ2). Používáme ji jen proto, že jinak by počet červených vrcholů na cestě podle ČČ1 mohl být o 1 větší než počet černých vrcholů - drobná komplikace.

Technicky vzato je ČČ1 zdaleka nejzávažnější, neboť její porušení se velmi obtížně napravuje, protože má globální charakter a týká se všech cest v celém stromě. Budeme se proto snažit ji stále dodržovat i za cenu dočasného porušení ČČ2, která se týká lokální situace a lépe se napravuje postupným přebarvováním vrcholů.

Ověřte si platnost všech podmínek a vyváženost stromu při postupném přidávání do stromu.

Podobně jako u AVL-stromů se vyhledávání, určování minima a maxima provádějí algoritmy binárního vyhledávacího stromu, které nemění tvar stromu ani obarvení vrcholů a proto nemohou porušit výše uvedené tři podmínky.

Scéna: *Rotace v červeno-černém stromu*

Následující scény ukáží, jak se provádí přidávání a vynechávání v červeno-černém stromu. Stejně tak jako u AVL-stromů je zde základní operací rotace, přibývá také přebarvování vrcholů.

V červeno-černém stromu budeme provádět rotaci hrany od vrcholu v k jeho synovi w v případě, kdy barva syna w je červená. Pokud je i barva vrcholu v červená, rotace barvy žádných vrcholů nezmění (tato možnost by správně neměla nastat, ale u některých operací povolíme na krátkou dobu porušení ČČ2). Pokud je barva vrcholu v černá, pak se barvy vrcholů prohodí. V animaci je to znázorněno tak, že se černá barva vrcholu v převede na vrchol w (a v tím zčervená).

Výše uvedená podmínka provádění rotace i způsob jejího provedení jsou vedeny tím, že za této situace se neporuší ČČ1. Pokud by se rotovala hrana, jejíž spodní vrchol je černý, k porušení ČČ1 by došlo. Zkuste si různé možnosti rotace v appletu.

I přípustná rotace ale může porušit ČČ2, pokud horní vrchol rotované hrany je černý a má dva červené syny. Jak ale bylo řečeno, porušení ČČ2 občas na chvíli tolerujeme, a proto takovou rotaci považujeme za přípustnou.

Scéna: *Červeno-černé vkládání - případ 0*

Nebudeme se zvlášť zabývat přidáním do prázdného stromu, kdy stačí vytvořit černý kořen jako jediný vrchol stromu a vložit do něj klíč.

U neprázdného stromu se, obdobně jako u AVL-stromů, i zde se nejprve přidá vrchol způsobem podle BVS algoritmu. V každém případě je přidán list vrcholu, kterému scházel syn. Nově přidaný vrchol obarvíme červeně. Tím se nemůže porušit ČČ1. Jelikož neměníme kořen stromu ani jeho barvu, neporuší se ani ČČ3.

V této scéně byl nový červený vrchol přidán jako syn k černému vrcholu a proto nedojde ani k porušení ČČ2 a proto po přidání vrcholu je možno skončit.

Scéna: *Červeno-černé vkládání - případ 1*

Podobně jako v předchozí scéně, i zde přidáme nový vrchol pomocí BVS algoritmu a obarvíme jej červeně. Je-li dán klíč, který má ležet v nově přidaném vrcholu, nemůžeme ovlivnit, kam bude nový vrchol přidán, a zde je přidán jako syn červeného vrcholu a proto přidáním dojde k porušení ČČ2. Proto musí následovat operace odstraňování dvou červených vrcholů nad sebou, kterou je obecně nutno provádět opakovaně.

Jak uvidíme dále, je výhodné si situaci popsat obecněji: ve stromu existuje vrchol v, který je červený a má červeného otce a kromě toho dvojice v a jeho otec je jediná dvojice porušující ČČ2. V dalším budeme ostatní vrcholy budeme označovat podle jejich vztahu k vrcholu v.

Při odstraňování dvou červených vrcholů nad sebou budeme rozlišovat tři případy. V této scéně tedy předpokládáme, že v a jeho otec jsou červení a navíc vrchol v má dědečka a červeného strýce.

Uvědomte si, že u právě popsané situace dvojice dědeček a otec vrcholu v je v pořádku a jelikož otec je červený, dědeček musí být *černý*.

Operace, kterou zde provedeme, je následující: dědečka přebarvíme na červeno a jeho syny (tedy otce a strýce vrcholu v - oba červené) začerníme. Operace je animována tak, že se černá barva z dědečka převádí na otce a strýce.

Uvedená operace, kterou budeme často provádět i v dalších scénách, zjevně neporuší platnost ČČ1. Navíc se otec vrcholu v začerní a tedy dvojice vrchol v - jeho otec již neporušuje ČČ2.

Provedení právě popsané operace však nutně nezaručí, že vznikne správný červeno-černý strom. Mohou nastat dva problémy.

Pokud dědeček vrcholu v měl červeného otce, vznikne nová nepřípustná dvojice červených vrcholů, přesně tak, jak se tomu stane v této scéně. Na tuto dvojici musíme znovu opakovat buď operaci popsanou v této scéně, kdy za vrchol v budeme nyní pokládat dědečka původního vrcholu v, nebo operace popsané dále, v závislosti na jejich aplikovatelnosti. V této scéně se zde popsaná operace - převedení černé barvy z dědečka aktuálního vrcholu v na jeho otce a strýce - aplikuje postupně třikrát. Jelikož však provedením této operace se problém v nejhorším případě převede o dvě patra výše, počet opakování nemůže být i v tom nejhorším případě větší než polovina výšky stromu.

Přebarvení dědečka vrcholu v na červeno může také způsobit jiný problém: porušení ČČ3 v okamžiku, kdy dědeček je kořen. Jak jsme ale již řekli, zde je náprava jednoduchá: zčervenalý kořen prostě začerníme a tím dostaneme správný červeno-černý strom a můžeme skončit. Takto také končí právě probíraná scéna.

Scéna: *Červeno-černé vkládání - případ 2*

Pokud ve stromu existuje právě jedna dvojice červených vrcholů, které jsou ve vztahu otec - syn (syna budeme zase označovat v), může nastat i jiná komplikace než ty, které byly popsána výše. V předchozí scéně jsme probrali dva případy: v nemá dědečka a nebo v má dědečka a červeného strýce. V této scéně a scéně následující probereme případ, kdy v má dědečka (který nutně musí být černý) a buď *nemá* strýce nebo je strýc *černý*. V obou případech budeme postupovat stejně, pro jednoduchost ukážeme jen postup v případě černého strýce.

Jistě si všimnete, že posledně jmenovaný případ nemůže nastat ihned po přidání nového vrcholu do červeno-černého stromu, protože černé skóre černého strýce by bylo větší než černé skóre nově přidaného vrcholu, který je list a tedy neúplný, viz ČČ1'.

V této scéně nejprve provedeme operaci z minulé scény, ale vrchol v, který při této operaci zčervená (bude to dědeček nově přidaného vrcholu) pak bude mít červeného otce, černého dědečka a černého strýce, nastane tedy případ 1. V této scéně navíc je vrchol v levým synem svého otce a jeho otec také levým synem svého otce, tj. dědečka vrcholu v. Obdobně by se postupovalo v zrcadlově symetrické situaci, kdy by jak v, tak i jeho otec byli praví synové.

V takovém případě provedeme rotaci hrany, spojující dědečka a otce vrcholu v. Jelikož je to hrana vedoucí od černého vrcholu dolů k červenému vrcholu, je rotace přípustná a nedojde proto k porušení ČČ1. Navíc má horní vrchol rotované hrany (dědeček vrcholu v) jednoho syna červeného (otec vrcholu v) a jednoho černého (strýc vrcholu v) a proto se nevytvoří nová hrana, porušující ČČ2. K vytvoření takové hrany by nedošlo ani pokud by vrchol v strýce neměl (případ, který zde neukazujeme).

Na druhé straně ale vidíme, že rotace začerní otce vrcholu v a proto hrana spojující otce vrcholu v s vrcholem v (která zůstává ve stromu i po rotaci) bude ČČ2 splňovat také. Provedenou operací tedy vznikne správný červeno-černý strom a můžeme skončit.

Scéna: *Červeno-černé vkládání - případ 3*

Tato scéna je obdobná jako předchozí: po přidání nového vrcholu vznikne nepřípustná červená hrana a po provedení spuštění černé barvy z dědečka nového vrcholu na dědečkovy syny se objeví vrchol v, který má červeného otce, černého dědečka a černého strýce.

Na rozdíl od předchozí scény ale je vrchol v pravým synem svého otce a jeho otec je naopak levým synem svého otce, tj. dědečka vrcholu v. Zrcadlově obráceně by se postupovalo, pokud by naopak byl v levým synem a jeho otec pravým synem.

Za této situace, označované jako Případ 3, provedeme nejprve rotaci hrany, spojující vrchol v s jeho otcem. Je to případ, o kterém jsem se zmiňoval ve scéně o rotaci - rotujeme hranu, která by ve správném červeno-černém stromu neměla být, ale dočasně ji připouštíme. Důležité je, že tato rotace je přípustná - neporuší ČČ1. Kromě toho zjevně rotovaná hrana zůstává jedinou hranou, porušující ČČ2. Povšimněte si také, že vrchol, který byl původně dědečkem vrcholu v je nyní jeho otcem.

Nyní je již vidět, proč jsme rotaci prováděli: převede strom na případ z minulé scény a proto stačí provést ještě rotaci hrany spojující vrchol v s jeho současným otcem (před chvílí ještě dědečkem) a dostáváme správný červeno-černý strom a operace končí.

Tím jsme probrali všechny situace, které mohou nastat při přidávání nového vrcholu do červeno-černého stromu.

Scéna: *Červeno-černé vkládání*

V této scéně si můžete zopakovat vkládání do červeno-černého stromu *ad libitum*. Při operacích s případnou hranou s červenými konci je vždy uvedeno, zda se jedná o 1., 2., nebo 3. případ podle předchozích scén.

Scéna: *Červeno-černé vynechávání - případ 1*

V této scéně a scénách následujících se budeme zabývat vynecháváním vrcholu z červeno-černého stromu. Vynechávání rozdělíme do 14 případů, z nichž každý má svou podmínku aplikovatelnosti a související akci. Při vynechávání najdeme první aplikovatelný případ (v pořadí, ve kterém jsou uváděny) a provedeme odpovídající akci. Při provádění akce některého případu tedy můžeme předpokládat nejen to, že je splněna jeho podmínka aplikovatelnosti případu, ale také, že *nejsou* splněny podmínky případů předchozích.

V dalším budeme vynechávaný vrchol značit v a označovat jej fialovým podkladem. Dále označíme (pokud existují) jeho otce jako u, jeho bratra jako w, levého syna bratra w jako x a pravého syna bratra w jako y. Opakuji, že počet černých vrcholů na cestě z kořene do nějakého vrcholu budeme nazývat *černé skóre* daného vrcholu.

Specifické vlastnosti červeno-černého stromu vedou k tomu, že někdy bude jednodušší provádět vynechávání jinak, než se dělá v klasickém BVS.

První scéna ukazuje vynechávání klíče úplného vrcholu, tedy vrcholu s oběma syny (v našem případě vynecháváme přímo kořen, ale to není podstatné). Zde budeme postupovat přesně stejně jako u BVS: zahodíme klíč, ale ponecháme vrchol, převedeme do něj nejblíže nižší klíč a pak vynecháme vrchol bez klíče který, jak již dávno víme, nemá pravého syna a proto se na něho vztahuje některý z následujících případů.

Akci si proveďte s tím, že vynechávání vrcholu bez pravého syna asi nebudete rozumět.

Scéna: *Červeno-černé vynechávání - případ 2*

V této scéně se vynechává klíč vrcholu v s jedním synem. V červeno-černém stromě tato situace nastává jen ve velmi speciálním případě. Ani syn, ani případný vnuk vrcholu v totiž nemůže být černý, protože v je neúplný, ale cesta do neúplného v by obsahovala méně černých vrcholů než cesta do jeho syna nebo vnuka. Syn vrcholu v tedy musí být červený. Kdyby existoval vnuk vrcholu v, pak jsme si právě řekli, že nemůže být černý, ale nemůže být ani červený jako syn červeného syna vrcholu v. v tedy nemá vnuky neboli jeho jediný syn je červený list.

Akce v tomto případě je následující (odlišná od BVS): klíč vrcholu v vyhodíme, přesuneme do něj klíč jeho syna a pak syna vynecháme. Jelikož syn vrcholu v je červený list, nezpůsobíme si tím žádné potíže.

Scéna: *Červeno-černé vynechávání - případ 3*

Další jednoduchý případ je vynechávání červeného listu. O tom jsme se již vlastně zmínili v závěru předchozí scény - prostě vrchol vynecháme a je to.

Scéna: *Červeno-černé vynechávání - případ 4*

V této a následujících scénách tedy budeme vynechávat vrchol, který je černým listem. Nemůžeme jej prostě vyhodit, protože by se tím porušila ČČ1 způsobem, ze kterého bychom těžko hledali nápravu a proto musíme postupovat daleko opatrněji a obecně také komplikovaněji.

Tato scéna je zatím zcela primitivní: předpokládáme, že vynechávaný černý list je zároveň kořenem. Vyhodíme jej a zůstane prázdný strom (což je matematická formulace pro případ, kdy nic nezůstane).

V dalších scénách tedy budeme předpokládat, že vynechávaný černý list v má otce u. Otec má *menší* černé skóre než jeho černý syn v a proto nemůže být neúplný. Vrchol v proto má i bratra, kterého budeme značit w.

Scéna: *Červeno-černé vynechávání - případ 5*

V této scéně budeme předpokládat, že bratr w vynechávaného černého listu v je *červený*. V takovém případě otec u vrcholu v je dle ČČ2 černý. Navíc černé skóre bratra w je menší než černé skóre vrcholu v a proto bratr w nemůže být neúplný. Jelikož bratr w je červený jeho dva synové musí být černí dle ČČ2. Není důležité, zda jsou to listy (jako je znázorněno) nebo mají syny.

V dané situaci provedeme rotaci hrany spojující černého otce u a červeného bratra w. Tato rotace je přípustná a neporušuje ČČ2, protože druhý syn otce u - vynechávaný vrchol v je černý. Dostaneme tedy správný červeno-černý strom, ve kterém vynecháváme vrchol v, který je stále černým listem, ale jehož nový bratr je jeho původní synovec (syn bratra w) a tedy je černý.

Rotace převede případ 5 na některý z případů následujících. Proveďte si animaci, i když akce po provedení rotace ještě nebyly vysvětleny.

Scéna: *Červeno-černé vynechávání - případ 6*

Od nynějška tedy budeme zkoumat situaci, kdy vynechávaný černý list v má otce u a *černého* bratra w.

V následujících dvou scénách budeme předpokládat, že otec u vynechávaného černého listu v je *černý* a jeho bratr *není* list. V této scéně předpokládáme, že v je levý syn a w pravý syn otce u a bratr w má levého syna x (není důležité, zda má i pravého syna). Zrcadlově symetricky bychom postupovali i pokud by v byl pravý syn a w levý syn, který by měl pravého syna (bez ohledu na to, zda by měl levého syna).

Povšimněte si, že černé skóre vrcholu v a jeho bratra w jsou stejná. Kdyby syn nebo případný vnuk bratra w byl černý, pak by měl černé skóre větší než je černé skóre samotného bratra w a tedy také větší než je černé skóre vrcholu v, což ale nemůže nastat, protože v je neúplný vrchol.

Levý syn x bratra w je tedy červený a jeho případný syn nemůže být ani červený (ČČ2) ani černý (viz předchozí odstavec). Syn x je tedy červený list.

Operace provedená v daném případě je jednoduchá, i když trochu zdlouhavá. Klíč vrcholu v vyhodíme, do v přesuneme klíč z jeho otce u, do otce u přesuneme klíč z levého syna x bratra w vrcholu v. Tím jsme se dostali do případu 3 - potřebujeme vyhodit červený list x a to jednoduše provedeme, aniž bychom ohrozili platnost podmínek červeno-černého stromu. Povšimněte si, že přesuny klíčů se provádějí tak, že žádný z nich nenaruší platnost pravidla o distribuci klíčů do vrcholů binárního vyhledávacího stromu.

Scéna: *Červeno-černé vynechávání - případ 7*

Tato scéna popisuje případ velmi podobný předchozí scéně. Vynechávaný vrchol v má černého otce a černého bratra, který není list. Na rozdíl od předchozí scény však bratr postrádá jednoho syna, toho, který by byl bližší vrcholu v. Na obrázku tedy bratr w má pouze pravého syna y. Stejně jako v předchozí scéně se dá dokázat, že synovec y musí být červený list.

Postupujeme podobně jako v minulé scéně, ale přesunů klíčů je více. Klíč vrcholu v vyhodíme, přesuneme do něj klíč otce u, do otce u přesuneme klíč bratra w a nakonec do bratra w přesuneme klíč synovce y. Pak již, podobně jako v předchozí scéně, vynecháme červený list y, který je nyní bez klíče. I zde je zřejmé, že přesuny klíčů nenarušují platnost pravidla o distribuci klíčů a vynechání červeného listu je vlastně převedení na případ 3.

Scéna: *Červeno-černé vynechávání - případ 8*

Pokud není aplikovatelný žádný z předchozích případů, pak jsou vynechávaný vrchol v i jeho bratr w černé listy. Zde si nejprve probereme několik jednoduchých případů. V této scéně předpokládáme, že společný otec těchto vrcholů je červený.

Operace, kterou provedeme nejdříve, bude často používána i v následujících scénách a je obrácením operace, kterou jsme prováděli při přidávání vrcholu. Ve stromu máme červený vrchol, který má dva černé syny. Černou barvu synů přesuneme na jejich společného otce. Touto operací se neporuší ani ČČ1, ani ČČ3. Pokud synové nemají žádného červeného syna (například pokud jsou oba listy), pak nemůže dojít ani k narušení ČČ2.

Právě popsanou operaci aplikujeme na otce u vynechávaného vrcholu a na jeho syny, tedy na vynechávaný vrchol samotný a jeho bratra. Vzpomeňte si, že zde vynechávaný vrchol a jeho bratr jsou (černé) listy. Po provedení

této operace se stane vynechávaný vrchol červeným listem a bez problémů jej vynecháme, viz případ 3.

Scéna: *Červeno-černé vynechávání - případ 9*

Poslední jednoduchý případ je když vynechávaný vrchol i jeho bratr jsou černé listy a jejich otec je černý a je kořenem stromu. Strom tedy má pouze tři vrcholy. Možná trochu složitě se vynechání provede tak, že se kořen přebarví na červeno (to poruší pouze ČČ3) a tím je případ převeden na situaci z předchozí scény. Tam popsané operace vynechají vrchol bez porušení ČČ1 a ČČ2 a platnost ČČ3 se automaticky obnoví.

Scéna: *Červeno-černé vynechávání - případ 10*

Nyní bude následovat několik případů, kdy prováděné akce jsou již značně komplikované. Ve všech z nich jsou vynechávaný vrchol v a jeho bratr w černé listy a jejich společný otec u je také černý a není kořen (vrchol v tedy má dědečka).

Zde i v dalších scénách týkajících se vynechávání si zavedeme pojem *dvojnásobně černý* vrchol. Bude to vrchol, který se do počítání černých vrcholů na cestách z kořene do neúplných vrcholů započítává dvakrát. Bude tedy například přípustné, aby (jak se tomu zakrátko stane při animaci) na některých cestách z kořene do neúplného vrcholu byly čtyři černé vrcholy a na jiných dva vrcholy černé a jeden dvojnásobně černý. Během provádění operací bude ve stromu obvykle nejvýše jeden dvojnásobně černý vrchol a jen velmi výjimečně dva. Dvojnásobně černý vrchol nebo vrcholy budou pochopitelně připuštěny jen dočasně a než vynechávání skončí se jich budeme muset zbavit.

Prvním krokem v této scéně i scénách následujících bude přenesení černé barvy vynechávaného vrcholu v a jeho bratra w do jejich společného otce u. Totéž jsme provedli i v případu 8, kdy ale otec byl červený a stal se tak jednoduše černý. Nyní byl otec černý a proto bude dvojnásobně černý. Druhou černou barvu budeme znázorňovat jako čtvercový černý podklad vrcholu.

Po přesunutí černé barvy z vynechávaného vrcholu a jeho bratra nahoru se vynechávaný vrchol stane červeným listem a bude možno jej snadno vynechat způsobem podle případu 3, ale než to uděláme, musíme se zbavit dvojnásobně černé barvy ve stromu. Ve všech následujících případech tedy bude naším úkolem zbavit se (jediného) dvojnásobně černého vrcholu, který budeme označovat V.

Nebudeme se zvlášť zabývat případem, kdy by byl dvojnásobně černý vrchol V kořenem, protože v tomto případě se prostě kořen změní na jednonásobně černý, což nezpůsobí žádný problém.

Dvojnásobně černý vrchol V tedy bude mít otce, kterého budeme označovat U. Bude mít i bratra, protože jinak by jeho otec byl neúplný vrchol a přitom by černé skóre jeho otce bylo menší než černé skóre vrcholu samého.

Bratr, který bude označován W, sám musí mít oba syny, protože jinak by byl neúplný a přitom je jeho černé skóre menší než černé skóre dvojnásobně černého vrcholu V.

První případ s dvojnásobně černým vrcholem je takový, že jinak správný červeno-černý strom obsahuje jeden dvojnásobně černý vrchol V, který má otce U a *červeného* bratra W. Protože bratr W je červený, jeho synové musí být černí. Otec je tedy černý, jinak by byla porušena ČČ2. Provedeme rotaci hrany spojující černého otce U s červeným bratrem W. Je to přípustná rotace, která (s uvážením černosti vrcholu V samotného) neporuší žádnou podmínku červeno-černého stromu. Bratrem vrcholu V se po provedení rotace stane jeden z dřívějších synů jeho původního bratra W, tedy černý vrchol a proto se dostaneme do situace, na kterou bude aplikován některý z dalších případů.

Proveďte si animaci operací, i když další postup ještě nebyl vysvětlen. Všimněte si jen, že po odstranění dvojitě černé barvy nakonec vynecháme požadovaný vrchol, který se na začátku stal červeným listem.

Scéna: *Červeno-černé vynechávání - případ 11*

Začínáme opět stejně, černou barvu dvou černých listů, z nichž jednoho chceme vynechat, přesuneme na jejich černého otce, který se stane dvojnásobně černým. Víme, že dvojnásobně černý vrchol V má otce U a bratra W a jelikož není aplikovatelný předchozí případ, bratr W je černý.

V této scéně předpokládáme, že společný otec je červený a bratr W nemá žádného červeného syna. Prosté převedení jedné černé barvy z dvojnásobně černého vrcholu V a černé barvy jeho bratra W na společného červeného otce U neporuší podmínky červeno-černého stromu a zbaví nás dvojitě černé barvy. Pak již jen vynecháme požadovaný vrchol, nyní červený list.

Scéna: *Červeno-černé vynechávání - případ 12*

Tento případ je podobný jako případ předchozí: ve stromě se nachází dvojnásobně černý vrchol V, který má černého bratra W, který nemá žádného červeného syna. Jediný rozdíl je v tom, že společný otec U vrcholu V a jeho bratra W je také černý.

Operace kterou provedeme bude stejná jako v předchozím případě, převedení černé barvy z V a W na U, ale rozdíl je v tom, že vrchol V bude po jejím provedení jednoduše černý, ale dvojnásobně černou barvu bude mít otec U. Proto budeme muset znovu provádět některý z případů 10-14, ale dvojnásobně černý vrchol se posunul o jedno patro směrem ke kořeni.

Viděli jsme, že případ 10 se převedl na případ 11 a každý z případů 11, 13 a 14 představuje krátkou konečnou posloupnost akcí, jejíž délka nezávisí na velikosti stromu, kterými se dvojnásobně černého vrcholu zbavíme, aniž bychom se znovu dostali do případu 12.

Jediný případ, kdy se provádění operací brzo nezastaví, je tedy případ 12, který vede opakovaně znovu na situaci podle případu 12. Jak ale bylo řečeno, poloha dvojnásobně černého vrcholu stále stoupá ke kořeni, takže se případ 12 nemůže opakovat vícekrát, než kolik je hloubka stromu. Nakonec se v nejhorším případě dostaneme do situace, kdy je nově vytvořený dvojnásobně černý vrchol kořenem a tomu prostě jednu černou barvu odejmeme, jak bylo popsáno výše.

Scéna: *Červeno-černé vynechávání - případ 13*

Případ 13 nastává, když se objeví dvojnásobně černý vrchol V, který má černého bratra W a ten má červeného syna Y na straně vzdálenější od vrcholu V, zatímco syn X bratra W, bližší k vrcholu V je černý. Buď je tedy V levý syn a W pravý syn jejich společného otce U a Y je pravý syn bratra W, nebo je V pravý syn a W levý syn a Y je levý syn bratra W. Na obrazovce je první možnost.

Barva společného otce U vrcholu V a jeho bratra W není důležitá. My případ 13 rozdělíme podle barvy otce U na 13a (otec černý) a 13b (otec červený). V obou případech ale budeme provádět stejné akce.

Podívejme se nejprve na 13a. Po vzniku dvojnásobně černého vrcholu se nejprve spustí černá barva bratra W na jeho syny. Od vrcholu V odvrácený červený synovec Y vrcholu V tím zčerná (jednoduše), zatímco přivrácený synovec X se stane druhým dvojnásobně černým vrcholem.

Zdálo by se, že se situace zhoršila, ale po rotaci hrany spojující černého otce U s nyní červeným bratrem W již jistě vidíte, že jsme blízko cíle. Dvojnásobně černý dřívější synovec X vrcholu V se nyní stal bratrem dvojnásobně černého vrcholu V a jejich společným otcem je vrchol U, který již byl předtím otcem vrcholu V a po provedení rotace dostal červenou barvu. Provedená rotace je přípustná, protože horní konec rotované hrany byl černý a dolní červený a navíc je vrchol V (dvojnásobně) černý a tedy se ani ČČ2 neporuší.

Nyní již stačí převést jednu červou barvu vrcholů V a X na otce U. Barva vrcholů V a X se sníží na jednoduše černou a vrchol U (jednoduše) zčerná a tím vznikne správný červeno-černý strom bez dvojnásobně černých vrcholů.

Nyní již stačí vynechat zvolený vrchol, nyní červený list a vynechávání je ukončeno.

V případě 13b je otec U červený a provádíme tytéž operace. Po spuštění černé barvy bratra W dolů však vznikne hrana spojující červeného otce U s nyní červeným bratrem W. Tuto hranu rotujeme stejně jako v případě 13a; i zde je rotace přípustná a nevytvoří jinou hranu porušující ČČ2. Nakonec posunutí jedné černé barvy z nově vzniklé bratrské dvojice V a X na U nejen odstraní dvojnásobně černé barvy vrcholů V a X, ale také začerněním jednoho z vrcholů do té doby ČČ2 porušující hrany spojující U a W obnoví platnost ČČ2.

Scéna: *Červeno-černé vynechávání - případ 14*

Nakonec nejkomplikovanější případ: ve stromu je dvojnásobně černý vrchol V, který má otce U a černého bratra W a tento bratr má červeného syna X na straně přivrácené k vrcholu V. Buď je tedy V levým synem a W pravým synem jejich společného otce U a X je levým synem vrcholu W (to je co vidíte na obrazovce) a nebo je zrcadlově obráceně V pravým a W levým synem a synovec X je pravým synem vrcholu W.

Barva druhého syna Y bratra W a barva otce U nejsou pro sekvenci prováděných operací podstatné; ukážeme si proto čtyři varianty 14a, 14b, 14c, 14d. V prvních dvou je otec U černý a v následujících dvou červený, zatímco syn Y je černý v první a třetí možnosti a červený ve zbývajících dvou.

Podívejme se nyní nejprve na 14a. Jakmile se objeví dvojnásobně černý vrchol V, provedeme rotaci hrany spojující jeho černého bratra s červeným synovcem X přivráceným k vrcholu V. Tím se situace převede na případ 13a a postupuje se, jak bylo ukázáno v minulé scéně.

Stejně postupujeme i v případě 14b. Zde se ale po úvodní rotaci objeví hrana s oběma konci červenými. Je to hrana, spojující původního bratra W se vzdáleným synovcem Y. V následujícím spuštění černé barvy z nového bratra vrcholu V, kterým je jeho původní synovec X, ale horní konec hrany porušující ČČ2 zčerná a platnost ČČ2 se obnoví. Pak již operace probíhají stejně jako v případě 14a.

Případy 14c a 14d probíhají analogicky případům 14a, respektive 14b s tím, že rotace je převede na případ 13b. V případě 14d tedy rotací vznikne jedna hrana porušující ČČ2 (spojuje původního bratra W s původním vzdáleným synovcem Y). Po spuštění černé barvy se tato hrana spraví, ale vznikne nová hrana porušující ČČ2 (spojuje původního otce U s původním bratrem W). Tato hrana je upravena při závěrečném přesunutí jedné z černých barev dvojnásobně černých vrcholů nahoru.

Scéna: *Operace na červeno-černém stromu*

V této scéně si můžete vyzkoušet libovolné operace na náhodně vytvořeném červeno-černém stromu předem zvolené velikosti.

Kapitola 3

B-strom

Podobně jako stromy z předchozí kapitoly, i B-strom se používá pro uchovávání množiny čísel, se kterou chceme provádět následující operace:

- **nalezení prvku** - je dáno číslo a má se zjistit, zda se v množině nachází a v kladném případě se také určí kde je uloženo (viz dále),
- **vložení prvku** - do množiny se vloží zadané číslo (pokud ovšem v množině již je, neprovede se nic),
- **vynechání prvku** - z množiny se vynechá číslo, zadané místem kde je uloženo (viz dále) - tím je zaručeno, že se v množině opravdu nachází; pokud nevíme, zda číslo, které chceme vynechat, v množině je a nebo kde je uloženo, musíme provést operaci nalezení prvku,
- **minimum, maximum** - naleznou nejmenší, resp. největší číslo, které v množině je.

Scéna: *Úvod*

Úvodní scéna je jen grafická ilustrace, knoflíkem **Další** přejděte na následující scénu.

Scéna: *B-strom*

Byl vytvořen B-strom o 25 vrcholech. Jedná se o strom, který má následující tři vlastnosti:

- pro nějaké přirozené číslo $k > 1$ (v tomto konkrétním případě je na začátku voleno $k = 3$) platí, že každý vrchol s výjimkou kořene má obecně alespoň k a nejvýše $2k$ synů,

- pokud strom má alespoň 3 vrcholy, má kořen alespoň 2 syny a nejvýše $2k$ synů a
- listy stromu se nacházejí všechny ve stejné hloubce pod kořenem.

Důvody pro zvláštní rozmezí počtu synů budou vysvětleny později.

Každý vrchol B-stromu obsahuje ℓ ukazatelů na syny (přičemž $k \leq \ell \leq 2k$) a mezi jednotlivými ukazateli je $\ell - 1$ datových buněk (tedy alespoň $k - 1$ a nejvýše $2k - 1$), každá buňka obsahuje jedno číslo.

Řekneme, že vrchol v je *následníkem* ukazatele p z některého vrcholu stromu, jestliže existují vrcholy $v_0, \ldots, v_k$ takové, že p ukazuje na vrchol v_0, pokud $k > 0$, pak pro $i = 1, \ldots, k$ některý ukazatel z vrcholu v_{i-1} ukazuje na v_i, a v_k.

Množinu s n prvky reprezentujeme B-stromem tak, že vytvoříme B-strom, který má ve svých vrcholech dohromady n datových buněk a do nich pak rozmístíme čísla množiny tak, aby platila následující pravidla:

- probíráme-li datové buňky jednoho vrcholu zleva doprava, čísla v nich uložená rostou,
- jestliže ukazatel p je v jistém vrcholu vlevo (resp. vpravo) od datové buňky b, pak všechna čísla ve všech datových buňkách vrcholů, které jsou následníky ukazatele p, jsou menší (resp. větší) než číslo v datové buňce b.

Čísla v datových buňkách jsou znázorněna graficky a nikoli číselně. Je vidět, že zleva doprava rostou a hodnoty synů jsou ve výše uvedeném vztahu s hodnotami otců. Klepnutím na datovou buňku libovolného vrcholu se hodnota v ní uložená zobrazí na ovladači vedle návěští **Hodnota** a současně se ve všech datových buňkách objeví červená vodorovná čára graficky odpovídající hodnotě ve zvolené datové buňce, aby bylo možné snadno určit, ve kterých datových buňkách jsou hodnoty menší a ve kterých větší.

Strom je v této scéně možno zvětšovat přidáváním náhodných čísel po jednom nebo po větším počtu a nebo je možné strom vymazat (a začít znovu přidávat).

Způsob zobrazení stromu je možno různě měnit. B-strom často vychází velmi široký a nevejde se na obrazovku. Je proto možné jej knoflíky ovladače zužovat a nebo naopak rozšiřovat a nebo nechat systém automaticky jeho šířku přizpůsobovat šířce obrazovky.

Podobně jako tomu bylo u binárního vyhledávacího stromu, operace se stromem obvykle probíhají podél cesty z kořene do některého listu stromu. Na ovladači je také možno zvolit, jak budou zobrazovány *neaktivní* vrcholy, neboli vrcholy, které při dané operaci neleží na cestě, kde probíhají úpravy. Volby jsou tři: neaktivní vrcholy se buď zobrazují stejně jako vrcholy aktivní nebo se zobrazují slabě na pozadí a nebo se nezobrazují vůbec.

První možnost ukazuje operaci zasazenu do celkového kontextu úplného stromu, ale pokud se šířka buněk volí tak, aby se celý (někdy velmi široký) strom vešel na obrazovku, musí být buňky velmi úzké, což snižuje přehlednost zobrazení. Poslední možnost ukazuje jen část stromu, ale tato část je cesta, která v každé vrstvě obsahuje obvykle jeden a málokdy více než dva vrcholy a proto i při malém rozlišení obrazovky je možno volit široké a přehledně zobrazené buňky a ukazatele ve vrcholech. Prostřední možnost je blízká poslední, ale nevýrazně naznačené neaktivní vrcholy upozorňují, že cesta je jen malá část stromu, ve kterém se pracuje.

Dále jsou dvě možnosti jak rozmístit vrcholy v rámci jedné vrstvy. Volba **Rozvinuté zobrazení** je rozmístí tak, že se navzájem nepřekrývají. Tato volba je prakticky povinná pokud jsou neaktivní vrcholy plně zobrazovány. Volba **Vertikální zobrazení** umístí každý vrchol středem pod střed ukazatele v jeho otci, který na něj ukazuje. Při této volbě se vrcholy v jedné vrstvě navzájem velmi překrývají a proto je tato volba nevhodná pokud se zobrazují i neaktivní vrcholy. Pokud jsou ale neaktivní vrcholy skryty nebo slabě nakresleny na pozadí (a tedy z každé vrstvy je plně nakreslen obvykle jediný vrchol aktivní cesty), pak je tato volba velmi výhodná, protože aktivní cesta je nakreslena zhruba svisle a proto je velmi snadné ji i u mimořádně velkého B-stromu přehledně znázornit na obrazovce.

Nakonec je možné zvolit, zda mají být vrcholy zobrazovány v plné šířce nebo v aktuální šířce. Při zobrazení v plné šířce je vrchol zobrazen v rámci, jehož šířka odpovídá $2k - 1$ datovým buňkám a $2k$ ukazatelům, ale pokud vrchol nemá maximální přípustný počet datových buněk, je využita jen část tohoto rámce. Při zobrazení v aktuální šířce odpovídá šířka nakresleného vrcholu aktuálnímu počtu jeho datových buněk a ukazatelů a může proto být až zhruba poloviční proti zobrazení v plné šířce. První možnost kreslí strom tak, že není nutné vrcholy přemisťovat například při přidání nebo ubrání datové buňky a odpovídajících ukazatelů, ale ve srovnání s druhou možností dosti plýtvá užitečnou plochou obrazovky.

Scéna: *Vyhledávání v B-stromu*

Podobně jako v binárním vyhledávacím stromu a jeho vyvážených variantách je i v B-stromu vyhledávání velmi jednoduché a je dáno pravidlem, jak se čísla ukládají do datových buněk vrcholů stromu. Provádí se tak, že opakujeme následující postup, vycházejíce přitom z kořene:
Hledáme-li číslo k a nacházíme-li se ve vrcholu v, pak nejprve prohledáme všechny datové buňky, které leží ve vrcholu. Pokud je v jedné z nich hledané číslo, bylo nalezeno a vyhledávání končí. V opačném případě jsou dvě možnosti. Buď je vrchol v v nejnižší vrstvě a v tom případě hledané číslo v žádné datové buňce stromu neleží, nebo přejdeme k synu vrcholu v tak, že je-li hledané číslo menší (resp. větší) než všechna čísla v datových buňkách vrcholu

v, pak přejdeme do syna, na kterého ukazuje levý (resp. pravý) ukazatel ve vrcholu v a pokud je hledané číslo mezi čísly v dvou sousedních datových buňkách vrcholu v, přejdeme na syna ukazovaného ukazatelem mezi těmito dvěma buňkami. Uvědomte si při tom, že čísla v datových buňkách jednoho vrcholu jsou uspořádána vzestupně.

Zvolte hledané číslo buď tak, že jej přímo zapíšete do příslušného pole na ovladači nebo klepnete na některou datovou buňku a v ní uložená hodnota se do pole ovladače zapíše automaticky (a může být případně dodatečně pozměněna). Pak proveďte výpočet standardním způsobem.

Scéna: *Hledání minima v B-stromu*

Hledání minima je velmi jednoduché. Vycházejíce z kořene, postupujeme stále do syna aktuálního vrcholu, užívajíce jeho levý ukazatel, dokud je tento nenulový. Když je tento postup zastaven, hledaná hodnota se nachází v levé datové buňce určeného vrcholu.

Scéna: *Hledání maxima v B-stromu*

Tato operace se provádí zrcadlově vzhledem k hledání minima. Vycházejíce z kořene, postupujeme stále do syna aktuálního vrcholu, užívajíce jeho pravý ukazatel, dokud je tento nenulový. Když je tento postup zastaven, hledaná hodnota se nachází v pravé datové buňce určeného vrcholu.

Scéna: *Štěpení vrcholu B-stromu*

Ukážeme nyní operaci, která se s výhodou používá při přidávání nové hodnoty do B-stromu. Každý vrchol stromu zobrazeného v této scéně má maximální počet datových buněk $2k-1$ (tedy 5, pokud máme $k = 3$). Při krokování knoflíkem **Krok** postupujeme podobně jako při vyhledávání jisté hodnoty, ale vrcholy pouze neprocházíme, ale každý navštívený vrchol "rozštěpíme".

Operace rozštěpení vrcholu v v prvním kroku $2k-1$ datových buněk (tedy 5, pokud je stále $k = 3$) rozdělí na tři skupiny. Levá z nich obsahuje levých $k-1$ (tedy 2 pro $k = 3$) datových buněk spolu s ukazateli, které s nimi sousedí. Pravá obsahuje pravých $k-1$ (tedy 2 pro $k = 3$) datových buněk se sousedícími ukazateli. Obě tyto skupiny tedy vytvářejí dva minimální vrcholy B-stromu. Poslední, prostřední, datová buňka vrcholu se osamostatní a obklopí se dvěma novými ukazateli, které ukazují na zbývající dva vrcholy, vytvořené rozštěpením původního vrcholu.

Pokud byl vrchol v kořenem stromu, pak nový malý vrchol vzniklý ze středové datové buňky vytvoří nový kořen stromu (který má minimální velikost, povolenou pro kořen). V souvislosti s tím pochopitelně vzroste počet vrstev stromu.

Pokud vrchol v nebyl kořenem, pak se v dalším kroku nový malý vrchol, vzniklý ze středové datové buňky a obsahující jednu datovou buňku obklopenou dvěma ukazateli, vsune do vrcholu, který je otcem vrcholu v, a to místo ukazatele, který ukazoval na vrchol v. Otci vrcholu v tím přibyla jedna datová buňka, ale jelikož v této scéně je otec pozůstatkem po štěpení vrcholu ve vyšší vrstvě, má minimální povolený počet datových buněk a proto v rámci limitu může další datovou buňku pojmout.

Štěpení je způsob, jak z vrcholu, který má maximální povolený počet datových buněk, vytvořit vrcholy, které mají minimální povolený počet buněk. Zde je také klíč k volbě vztahu minimálního a maximálního počtu ukazatelů a datových buněk vrcholu: pro datové buňky je dvakrát minimum plus jedna rovno maximu.

Scéna: *Přidávání nové hodnoty do B-stromu*

Základní myšlenka přidání nové hodnoty je velmi jednoduchá: postupujeme jako kdybychom přidávanou hodnotu hledali. Pokud ji nalezneme, již ve stromu je a není třeba ji tam přidávat. (Tento případ by u dobře navrženého algoritmu neměl nastat.) Když není nalezena, pak postup skončí v některém vrcholu v nejnižší vrstvy, kde by se hodnota měla nacházet, ale není tam. Vrchol přitom nemá syny a tudíž hledání dále nepostupuje. Do vrcholu v proto přidáme novou datovou buňku a do ní vložíme přidávané číslo. Pravidlo o uspořádání čísel v datových buněk jednoznačně určuje, kam je třeba datovou buňku přidat. Současně s přidáním jedné datové buňky je také třeba přidat jeden ukazatel na syna, který pochopitelně má nulovou hodnotu, protože vrchol v nejnižší vrstvě nemá syny.

Potíž ale nastává, pokud uvažovaný vrchol v již má maximální povolený počet datových buněk. Některé varianty B-stromu pracují tak, že v tomto případě vrchol rozštěpí a přidají novou datovou buňku do patřičného z vzniklých (minimálních vrcholů). Pokud ovšem otec vrcholu v (a popřípadě i jeho vzdálenější předchůdci) byl také maximální, nebyl by schopen přijmout novou datovou buňku a proto je nutné štěpení provádět i ve vyšších úrovních, někdy až do kořene stromu.

Algovize ale ukazuje variantu B-stromu, která může být označena jako "opatrná". Při přidávání postupujeme stromem z kořene dolů do listu, jako bychom přidávanou hodnotu hledali; pokud ale přitom narazíme na vrchol s maximálním počtem datových buněk, pro jistotu jej *ihned* rozštěpíme. Štěpení v jedné vrstvě zaručí, že je bez problémů proveditelné štěpení ve vrstvě, která je pod ní (otec štěpeného vrcholu bude schopen absorbovat novou datovou buňku) a když dospějeme až do nejnižší vrstvy, bude vrchol připraven k absorpci nové buňky pro přidaný vrchol.

Zkuste si přidávání provádět nebo krokovat. Snažte se přitom přidávané hodnoty volit tak, aby co nejčastěji docházelo ke štěpení vrcholů.

Scéna: *Jednoduché vynechávání z B-stromu*

Ve zbývajících scénách kapitoly o B-stromech je naším cílem vynechat jistou hodnotu z B-stromu. Animace ve scénách předpokládá, že nevíme, ve které datové buňce je hodnota uložena a proto se na začátku operace provede vyhledání této datové buňky; běžec ukazující průběh operace po vstupu do této datové buňky zčervená. (Pokud by se zjistilo, že hodnota ve stromě uložena není, operace se ukončí ihned poté. V dobře navrženém algoritmu využívajícím B-stromu by ale tato možnost neměla nastat.)

V následujícím výkladu budeme používat pojem levý bratr a pravý bratr vrcholu. Jsou to vrcholy, ležící ve stejné vodorovné vrstvě jako uvažovaný vrchol a to bezprostředně vlevo, resp. vpravo od něho (pokud ovšem existují). Formálně je lze definovat takto: nechť je dána datová buňka ve vrcholu, který není v nejnižší vrstvě a L, resp. P jsou ukazatelé, kteří s touto datovou buňkou sousedí zleva, resp. zprava. Nechť v_L, resp. v_P jsou vrcholy, na které ukazuje ukazatel L, resp. P. Pak v_L je levý bratr vrcholu v_P a v_P je pravý bratr vrcholu v_L.

Vynecháváním se bude zabývat kromě této scény i pět následujících scén. Zde se budeme zabývat jednoduchým případem, ve kterém se tvar B-stromu (skoro) nezmění. Datová buňka s vynechávanou buňkou nachází v nejnižší hladině ve vrcholu, který má více než minimální povolený počet datových buněk. Navíc předpokládáme, že B-strom má více než jeden vrchol, takže má alespoň 2 vrstvy.

V tomto případě se prostě datová buňka a (nulový) ukazatel s ní zleva sousedící vynechají. Případné buňky a ukazatele, které byly vpravo od vynechané buňky, se musí posunout vlevo, aby ve vrcholu nevznikla "díra".

Scéna: *Vynechávání z B-stromu s jedním vrcholem*

Než se pustíme do výkladu složitějších případů, rozebereme v této scéně jeden triviální případ: B-strom má jediný vrchol (který je pochopitelně kořenem).

Pokud kořen má více než jednu datovou buňku, pak požadovanou buňku můžeme prostě vynechat spolu s ukazatelem ležícím nalevo od ní; kořen je výjimka a může obsahovat jedinou datovou buňku obklopenou dvěma ukazateli. Pochopitelně se v případě potřeby zbylé musí buňky a ukazatele posunout vlevo, aby nevznikla díra.

V opačném případě je vynechávaná buňka jedinou datovou buňkou stromu a po jejím vynechání je B-strom prázdný. (V naší implementaci je prázdný strom představován jedním vrcholem, obsahujícím jediný nulový ukazatel a žádnou datovou buňku.)

Vynechávejte z kořene buňky tak dlouho, dokud ve stromu žádná nezbude. B-strom pak můžete obnovit stisknutím knoflíku **Nový strom** a vynechávání z osamoceného kořene opakovat.

V následujících scénách budeme předpokládat, že strom má alespoň dva vrcholy a tedy i alespoň dvě vrstvy.

Scéna: *Vynechávání s nepřímou adopcí zleva*

V této scéně se vynechávaná datová buňka se nachází v nejnižší hladině ve vrcholu v, který má minimální počet datových buněk, ale vrchol má levého bratra, která má větší než minimální počet datových buněk. Činnost potřebná pro vynechání buňky je v krokování rozdělena do 5 kroků.

Krok: *Vynechání buňky*

V prvních dvou krocích se vynechá vynechávaná hodnota i s datovou buňkou, která ji obsahovala a ukazatelem napravo od ní. Vznikne tím vrchol, který má nepřípustně malý počet buněk. (Po prvním kroku je vrchol nakreslen s prázdným místem po vynechané buňce, v druhém se stáhne na novou aktuální šířku.) ◇

Krok: *Přesun buňky z otce*

Nechť P je ukazatel v otci vrcholu v, který na vrchol v ukazuje a B je buňka vlevo od tohoto ukazatele v otci vrcholu v. (Tato buňka musí existovat, protože v má levého bratra - je to buňka mezi ukazateli na v a jeho levého bratra.) V následujících dvou krocích se nalevo ve vrcholu v vytvoří místo pro nový ukazatel a datovou buňku a pak se do nového místa pro buňku přesune z otce buňka B. Tím je ve vrcholu v dosaženo přípustného minimálního počtu datových buněk (ale ve vrcholu zatím schází nejlevější ukazatel). ◇

Krok: *Přesun buňky z levého bratra do otce*

Poté se nejpravější datová buňka levého bratra vrcholu v přesune na místo, kde byla původně buňka B a nejpravější ukazatel levého bratra vrcholu v se přesune na volné místo pro ukazatel na levém okraji vrcholu v. Tím je operace dokončena. Levému bratru vrcholu v ubyla jedna datová buňka, ale podle našich předpokladů je to přípustné. ◇

Scéna: *Vynechávání s nepřímou adopcí zprava*

Vynechávaná datová buňka leží ve vrcholu v v nejnižší vrstvě, vrchol v má minimální počet datových buněk a buď nemá levého bratra a nebo levý bratr vrcholu v má minimální počet datových buněk, ale vrchol v má pravého bratra, který má větší než minimální počet datových buněk.

V tomto případě operace probíhá v zásadě zrcadlově symetricky s předchozím scénou: po vynechání buňky se na pravou stranu vrcholu v přidá z otce buňka napravo od ukazatele, ukazujícího na v a pak se do uprázdněného místa v otci přesune levá buňka pravého bratra vrcholu v. Kromě toho levý ukazatel z pravého bratra se přesune do vrcholu v a navíc se všechny buňky a

ukazatelé, zbývající v pravém bratrovi musí posunout doleva, aby se zaplnilo uprázdněné místo.

Scéna: *Vynechávání spojením vrcholu s bratrem*

Tato scéna popisuje případ, kdy nelze aplikovat adopci. Datová buňka leží ve vrcholu v v nejnižší vrstvě, má minimální počet buněk, levý bratr buď neexistuje nebo má minimální počet buněk a pravý bratr buď neexistuje nebo má minimální počet buněk. Je zřejmé, že vrchol v nemůže být současně bez levého i pravého bratra. Rozebereme případ, kdy má levého bratra, případ kdy má (jenom) pravého bratra se provede zrcadlově symetricky.

Krok: *Vynechání buňky*

V tomto případě se nejprve vynechá buňka s vynechávanou hodnotou; vznikne vrchol s nepřípustně malým počtem buněk, rovným $k - 2$. ◇

Krok: *Přesun buňky z otce*

Dále se buňka ležící v otci vrcholu v mezi ukazateli ukazujícími na levého bratra vrcholu v a vrchol v přesune jako nová pravá buňka levého bratra vrcholu v. Levý bratr vrcholu v pak má $(k - 1) + 1 = k$ buněk. ◇

Krok: *Vlastní spojení*

Potom se všechny buňky a ukazatele vrcholu v přenesou se zachováním pořadí napravo do levého souseda vrcholu v, který potom bude mít $2k - 2$ buněk, tedy přípustné množství. ◇

Krok: *Vypuštění ukazatele*

Nakonec se v otci vrcholu v vypustí ukazatel, který původně ukazoval na vrchol v (který již nyní nic neobsahuje) a buňky tohoto vrcholu se srazí dohromady. ◇

Krok: *Zpracování otce*

Otec vrcholu v má nyní o jednu datovou buňku méně. Pokud má stále přípustné množství buněk, operaci je možno ukončit. V opačném případě jsou dvě možnosti.

- Vynechání kořene: Pokud otec vrcholu v byl kořen, neobsahuje nyní žádnou datovou buňku a je tvořen jediným ukazatelem, který ukazuje na vrchol vzniklý spojením vrcholu v a jeho levého bratra. Pak prostě vypustíme kořen stromu a za nový kořen prohlásíme vrchol vzniklý spojením vrcholu v a jeho levého bratra. Toto je jediný případ, kdy poklesne počet vrstev B-stromu.

- *Otec w vrcholu v nebyl kořen:* Pak opakujeme výše uvedený postup na vrchol w: jestliže w má levého bratra s větším než minimálním počtem buněk, pak w přibere jednu buňku od otce a tu nahradíme pravou buňkou levého bratra vrcholu w, jinak pokud má w pravého bratra větší než minimální velikosti, aplikujeme zrcadlový postup; nakonec pokud vrchol w nemá ani levého ani pravého bratra větší než minimální velikosti, sloučíme vrchol w s jedním jeho bratrem a jednou buňkou z otce vrcholu w a celý postup popřípadě iterujeme pro otce vrcholu w.

◇

Scéna: *Vynechávání hodnoty z buňky ve vyšší vrstvě*

Poslední možnost při vynechávání je případ, kdy máme vynechat hodnotu ležící v buňce B ve vrcholu v, který není v nejnižší vrstvě. V tomto případě si pomůžeme podobným trikem, jako při vynechávání vrcholu s dvěma syny v binárním vyhledávacím stromu:

Krok: *Odstranění hodnoty z buňky*

Hodnotu uloženou v buňce B vyhodíme; buňka je dočasně prázdná. ◇

Krok: *Nalezení náhradní buňky*

Nalezneme buňku stromu s hodnotou nejblíže nižší k hodnotě uložené ve vynechávané buňce. Tuto buňku nalezneme tak, že se nejprve spustíme o jednu vrstvu dolů do vrcholu, ukazovaného ukazatelem ležícím bezprostředně vlevo od buňky B ve vrcholu v a pak postupujeme stále pravým ukazatelem aktuálního vrcholu, až se dostaneme do vrcholu nejnižší vrstvy a v něm si vybereme pravou buňku. ◇

Krok: *Přesun hodnoty z náhradní buňky*

Číslo uložené v buňce nalezené v předchozím kroku přesuneme do nyní prázdné buňky B. ◇

Krok: *Vynechání buňky*

Prázdnou buňku ze spodní vrstvy vynecháme. Tato buňka leží ve vrcholu nejnižší vrstvy a proto lze aplikovat některý z postupů popsaných v předchozích scénách. ◇

Scéna: *Operace s B-stromem*

Vyzkoušejte si nyní operace s B-stromem. Máte možnost si nechat vytvořit náhodný B-strom libovolné rozumné velikosti a s ním provádět operace dle vlastního výběru. Uvědomte si, že stupeň rozvětvení vrcholu (neboli číslo k) je v appletu pro názornost poměrně malé, ale v praktických aplikacích může

být několik desítek. Pak je i hloubka ohromných stromů velmi malá. B-stromy se používají často v situaci, kdy velká část stromu je uložena v pomalé paměti (např. disk) a není problém v rychlé operační paměti několik velmi velkých vrcholů, ale je třeba minimalizovat počet vrcholů, se kterými se setkáme na cestě ke kořenu, tedy se snažíme o co nejmenší hloubku stromu.

Kapitola 4

Halda

Halda je datová struktura, která umožňuje provádět s množinou čísel efektivně následující operace:

- **určení minima** - je-li halda neprázdná, pak se určí velikost minimálního prvku v haldě;
- **přidání nového čísla do haldy** - do haldy se vloží zadaný prvek (musí být zaručeno, že ještě v haldě není, halda test neumožňuje provádět);
- **vynechání minima** - je-li halda neprázdná, vynechá se nejmenší prvek, který v haldě je;
- **vynechání obecného prvku množiny** - s určenou polohou v haldě; halda neumožňuje vynechávání obecného prvku zadaného hodnotou.

Halda neumožňuje jednoduchým a efektivním způsobem určit maximální prvek a zjistit, zda (a kde) v ní leží zadaný prvek. Uvidíme, že toto omezení umožní použít velmi jednoduchou strukturu.

Přímočaré použití haldy je pro třídění: nejprve do haldy tříděná čísla "nasypeme" operací přidání a potom je vytahujeme jedno po druhém v setříděném pořadí operací vynechání minima. Některé algoritmy tento postup provádějí on-line a občas zjistí, že některý přidaný prvek do souboru ve skutečnosti nepatří, proto je pak výhodná i operace vynechání obecného prvku. Zjistit minimum je mnohdy užitečné i když jej nemíníme okamžitě vyjmout. Operaci vynechání minima by bylo možno zkombinovat z vyhledání minima a obecného vynechání, ale uvádíme ji samostatně, protože se jednak u haldy používá častěji než obecné vyhledání, a kromě toho je jednodušší.

Všechny haldy se implementují zakořeněným stromem nebo někdy lesem, který má stejný počet vrcholů jako je prvků representované množiny a do kaž-

dého vrcholu se jeden prvek množiny vloží. Prvek uložený ve vrcholu budeme nazývat *klíč* vrcholu.

U haldy vždy vyžadujeme, aby klíč jakéhokoli vrcholu *nebyl větší* než klíč jeho libovolného syna. To vede k tomu, že minimum je vždy uloženo v kořenu stromu nebo kořenu některého stromu lesa a je tedy snadné jej najít a často je i jednodušší vynechat kořen než obecný vrchol stromu.

V této kapitole popíšeme jednoduchou implementaci haldy pomocí stromu velmi speciálního tvaru. Pokud někdy uslyšíte slovo "halda" bez bližšího upřesnění, nejspíše je míněna takto implementovaná halda.

Scéna: *Tvar haldy*

Halda popisovaná v této kapitole je tvořena binárním stromem, splňujícím následující podmínky:

- každý vrchol má 0, 1 nebo 2 syny. Pokud má dva syny, označujeme je jako levý a pravý, pokud má jediného syna, je tento syn levý;
- pouze v nejnižších dvou vrstvách mohou být vrcholy, které mají méně než dva syny (v nejnižší vrstvě pochopitelně jsou pouze listy, tedy vrcholy bez synů);
- postupujeme-li předposlední vrstvou zleva doprava, pak nacházíme nejprve vrcholy se dvěma syny (tato skupina ale nemusí vůbec být přítomna), pak případný jediný vrchol s jedním (levým) synem a pak popřípadě vrcholy bez synů (listy).

Na počátku je v této scéně vidět prázdná obrazovka - halda je prázdná. Klepejte na knoflík **Přidej vrchol.** Každé klepnutí přidá jeden vrchol; klíče volíme pro jednoduchost postupně $1, 2, 3, \ldots$. Tento postup zajistí, že klíče budou splňovat podmínku haldy, aniž bychom museli provádět operace, které budou popisovány v dalších scénách.

Zajisté vidíte, že chceme-li splnit výše uvedené podmínky, pak tvar naší haldy je jednoznačně dán počtem jejích vrcholů - je pouze jediné místo, kam lze zavěsit nově přidávaný vrchol:

- pokud všechny vrcholy předposlední vrstvy mají 2 syny a nebo pokud tato vrstva schází (tj. halda obsahuje jen kořen), připojí se nový vrchol jako levý syn nejlevějšímu vrcholu *spodní* vrstvy;
- pokud je v předposlední vrstvě vrchol s jedním synem (takový vrchol může být jen jeden a syn je levý), připojí se nový vrchol jako pravý syn uvedeného vrcholu s jedním synem;
- pokud předposlední vrstva existuje a obsahuje alespoň jeden list, ale žádný vrchol s jedním synem, připojí se nový vrchol jako levý syn nejlevějšího listu v předposlední vrstvě.

Přidávejte vrcholy dokud není halda dosti velká a dokud vám není zcela zřejmé, jak halda musí vypadat.

Z haldy je též možno ubírat vrcholy knoflíkem **Uber vrchol.** Pro zachování tvaru haldy musí být vždy odebrán nejpravější vrchol nejnižší vrstvy. Je též možno přidávat nebo ubírat více vrcholů najednou, změní-li se číslo v poli vedle $\Delta =$**.**

Scéna: *Přidávání do haldy*

Přidávání obecného klíče do haldy se provede takto:

Nejprve se do haldy přidá nový vrchol do místa ve stromu, které bylo popsáno v předchozí scéně a pak se do nového, zatím prázdného, vrcholu v přidá nový klíč. Vrchol v ale může porušovat podmínku haldy, neboť klíč v něm obsažený může být *menší* než klíč jeho otce. V takovém případě necháme klíč z vrcholu v "vybublat" až do vrstvy, do které náleží: prohodíme klíč vrcholu v a jeho otce. Vrchol v je nyní už v pořádku, ale jeho otci se klíč zmenšil a zase může haldovou podmínku porušovat on; pak bychom prohodili klíč otce a dědečka vrcholu v; pak ovšem musíme kontrolovat haldovou podmínku pro dědečka, atd. Může se stát, že přidaný klíč vystoupá až do kořene (pokud byl novým minimem haldy).

Scéna: *Vynechávání minima*

Z podmínky pro ukládání klíčů do vrcholů haldy je zřejmé, že minimum haldy se vždy nachází v jejím kořenu. Máme-li tedy minimum vynechat, nemůžeme jej odstranit společně s vrcholem, ve kterém leží, protože by se nám strom rozpadl těžko opravitelným způsobem.

Pomůžeme si proto trikem známým z předchozích kapitol. Klíč z kořene odstraníme, ale vrchol ponecháme. Pak do něj přeneseme klíč jiného vrcholu a vynecháme pak vrchol, který se ocitne bez klíče.

Aby vynechání vrcholu, kterému odebereme klíč, nepokazilo tvar haldy, musíme za něj zvolit nejpravější list spodní vrstvy. Tím ale nastane obvykle problém: klíč tohoto vrcholu, který je až ve spodní části haldy, je většinou hodně velký a přeneseme-li jej do kořene, nebude se tam hodit, protože bude porušovat haldovou podmínku. Tento problém vyřešíme tím, že pak klíč necháme "spadnout" tak hluboko, až se dostane do místa, kde už nebude působit potíže.

Bude to operace v jistém smyslu opačná k "probublávání" nahoru při přidávání. Pokud klíč kořene je větší než klíč jeho syna, pak je prohodíme. Pokud ale má halda alespoň 3 vrcholy, má kořen dva syny a není jedno, se kterým klíč prohodit. Zásadně musíme prohození provést tak, že prohazujeme se synem, který má menší klíč; v opačném případě bychom porušení podmínky neodstranili (víte proč?).

Prohozením se klíč z kořene dostane o jednu hladinu níže, ale stále může porušovat haldovou podmínku (nyní s vrcholem, který je vnukem kořene). V takovém případě zase problémový klíč prohodíme s klíčem toho syna aktuálního vrcholu, který má menší klíč z obou bratrů. Prohazování klíčů se může obecně opakovat mnohokrát, ale nakonec klíč padající z kořene skončí buď v listu nebo vrcholu, jehož syn nebo synové mají klíče větší.

Scéna: *Vynechávání z haldy*

Vynechávání klíče z obecného vrcholu haldy se provádí velmi podobně jako vynechávání minimálního klíče z kořene. Máme-li vynechat klíč vrcholu v, pak klíč z v vyhodíme, do v převedeme klíč z nejpravějšího vrcholu spodní vrstvy haldy (pokud ovšem v není tímto vrcholem) a pak nejpravější vrchol spodní vrstvy haldy prostě odtrhneme.

Klíč, který se nyní octnul ve vrcholu v, může porušovat haldovou podmínku dvojím způsobem: buď je moc malý - menší než klíč otce vrcholu v - nebo je moc velký - větší než klíč některého z obecně dvou synů vrcholu v. Pokud dojde k prvnímu případu, necháme klíč vybublat vzhůru, tak jak jsme to dělali při přidávání. V druhé případě zase klíč necháme spadnout do patřičné vrstvy stejně jako tomu bylo při vynechávání minima.

Klepnutím zvolte některý vrchol haldy a krokujte si operaci vynechávání klíče.

Scéna: *Změna klíče*

Klepněte na některý vrchol; jeho klíč se objeví v poli napravo od $N =$**.** Nyní do tohoto pole napište jiné číslo. Tím se také změní klíč zvoleného vrcholu. Pokud změněný klíč porušuje haldovou podmínku, necháme jej vybublat nahoru, pokud je příliš malý nebo naopak spadnout dolů, pokud je příliš velká. Operaci provádíme přesně tak, jak bylo popisováno v předchozích scénách pro přidávání a vynechávání.

Změna klíče se obvykle nepovažuje za základní haldovou operaci, ale často je užitečná sama o sobě a především se jedná o základní netriviální součást všech výše uvedených operací.

Scéna: *Operace s haldou*

V této scéně se nedozvíte nic nového, ale můžete si zkusit libovolné operace s haldou dle svého uvážení. Obrázky haldy v knihách jsou obvykle malé, zde si ale můžete zkusit vytvořit opravdu velkou haldu, abyste viděli, že i při velkém počtu vrcholů je její hloubka stále malá, ale roste rychle do šířky, což ovšem nevadí, protože rychlost výpočtu určuje hloubka a nikoli šířka haldy. Kdykoli si také můžete haldu nechat vytvořit znova; její velikost si můžete nastavit na ovladači.

Scéna: *Číslování vrcholů haldy*

V této scéně je na obrazovce zobrazena velká halda, klíče vrcholů zobrazeny nejsou, protože pro následující úvahu nejsou nutné. Vrcholy haldy jsou ale očíslovány čísly $1, 2, \ldots$ tak, že jsou číslovány po vrstvách shora dolů a v rámci každé vrstvy zleva doprava. Z první scény již víme, že takto by vypadaly klíče haldy, do které bychom je přidávali v pořadí $1, 2, 3, 4, \ldots$.

Je vidět, že (pokud existují) má kořen číslo 1, jeho synové mají čísla 2, 3, vrcholy další vrstvy čísla 4, 5, 6, 7 atd.

Nejdůležitější pozorování je ale následující: pokud má vrchol pořadové číslo k, pak jeho levý syn (existuje-li) má pořadové číslo $2k$ a jeho pravý syn (existuje-li) má pořadové číslo $2k + 1$. Platnost tohoto tvrzení si ověřte si na obrazovce pro všechny zobrazené vrcholy. Možná přijdete na poměrně jednoduchý důkaz tvrzení sami.

Naopak tedy pokud vrchol jiný než kořen má pořadové číslo ℓ, pak jeho otec má pořadové číslo $\lfloor \ell/2 \rfloor$ (nebo $\ell/2$ pokud '/' chápeme jako operátor celočíselného dělení).

Další vlastností očíslování je, že považujeme-li vrstvu obsahující kořen za nultou, pak nejlevější vrchol k-té vrstvy má pořadové číslo 2^k a nejpravější vrchol této vrstvy má pořadové číslo $2^{k+1} - 1$.

Volbu na ovladači nyní změňte z **Normální** na **Zkosená.** Halda se nakloní tak, že x-ová souřadnice vrcholu je úměrná jeho výše uvedenému očíslování. Nyní změňte volbu na **Halda a pole.** Pod haldou se objeví jednorozměrné pole, pod každým vrcholem jedna jeho položka. Pod položkami jsou ještě jednou uvedena pořadová čísla, shodná s čísly vrcholů nad nimi.

V poli je implicitním způsobem zahrnuta struktura stromu, vytvářejícího haldu. Položka s pořadovým číslem k má za otce položku s pořadovým číslem $\lfloor k/2 \rfloor$ a levého a pravého syna s pořadovými čísly $2k$ a $2k + 1$, pokud tyto položky jsou v poli zahrnuty.

S haldou tedy je možno počítat tak, že ji máme representovanou jako pole a na rozdíl od velké většiny jiných datových struktur užívajících stromy nepoužíváme explicitně vyjádřené ukazatele na otce a syny, bratry a pod.

Pokud změníme volbu na ovladači na **Samotné pole,** pak ze scény zmizí zobrazení stromu a zůstane pouze pole. Operace se nám pak jeví tak, jak jsou obvykle naprogramovány pro počítač a strom, který nám umožňuje metodám lépe rozumět, již zůstává jen v naší mysli.

Scéna: *Operace s haldou ve tvaru pole*

Tato scéna má plnou funkčnost standardní haldy a umožňuje přidávat a vynechávat i určovat minimum. Volbou na ovladači můžeme zvolit libovolné ze čtyř zobrazení: normální a zkosená halda, zkosená halda s polem a samotné pole a sledovat provádění operací tak, jak se pozvolna transformuje z matematické formy do podoby programátorské. V kapitole o třídění posloupnosti

čísel algoritmem Heapsort si vše zopakujeme v základní aplikaci, ze které halda vznikla.

Kapitola 5

Slučovatelná halda

Slučovatelná halda je především halda, to znamená datová struktura, která umožňuje provádět efektivně operace určení minima, vkládání a vynechávání minima i obecného prvku, které jsme popsali v předchozí kapitole. Umožňuje ale také efektivně provádět operaci *sloučení*, což je sjednocení dvou *disjunktních* hald do jedné. Požadavek disjunktnosti umožňuje, že se množiny, popsané slučovanými haldami položí vedle sebe a upraví se jejich vnitřní struktura při zachování množiny vrcholů, aniž by mohlo dojít k vícenásobnému výskytu některého prvku.

Scéna: *Binomická halda*

V této kapitole se budeme zabývat jednou implementací slučovatelné haldy, která se nazývá binomická halda. Binomická halda nevyužívá jediný strom, ale les - soubor několika stromů. Tvar lesa, který vytváří binomickou haldu, je jednoznačně dán počtem jeho vrcholů. Pro reprezentaci množiny o n prvcích vytvoříme les, který má dohromady n vrcholů a do nich vložíme prvky množiny, které zde opět budeme nazývat *klíče* a to tak, aby klíč libovolného vrcholu, který není kořenem některého lesa, byl *větší* než klíč jeho otce.

Tato úvodní scéna je ilustrací toho, jak binomická halda vypadá pro různé počty vrcholů; jak funguje si ukážeme až v dalších scénách. Vidíte, že její stromy mají velmi různé velikosti; možná jste si již všimli, že povolené velikosti stromů jsou mocniny dvojky: 1, 2, 4, 8, 16, atd. a od každé velikosti je v haldě nejvýše jeden. Stromy přitom i při velké velikost mají malou výšku a je jich málo. Jak uvidíme později, počet stromů i hloubka nejhlubšího stromu v haldě o n vrcholech je nejvýše $\log_2 n$.

Měňte číslo v poli vedle $N =$ a poté klikněte na knoflík **Nová halda** pro vytvoření hald různé velikosti a sledujte jejich tvar. V této scéně nelze zatím provádět žádné operace, její cíl je ukázat, jak binomické haldy různé velikost vypadají. Zajímavé je i použití knoflíku **Přidej vrchol,** který haldu zvětší

o jeden vrchol. Pozornější z čtenářů jistě napadne řada vlastností binomické haldy a souvisejících operací, které budou v dalším textu probírány.

Scéna: *Stromy binomické haldy*

Klíčem k pochopení binomické haldy je dobrá znalost vlastností stromů, ze kterých se skládá. Proto se tato scéna jejich konstrukcí bude zabývat. Stromy binomické haldy mají velmi specifický tvar. Počet vrcholů takového stromu je vždy mocnina dvojky a má-li takový strom 2^k vrcholů, pak jeho kořen má k synů a strom má hloubku k. Takový strom budeme nazývat *strom řádu k*.

Konstrukce stromu, který se může objevit v binomické haldě, je jednoduchá:

- strom, který má $2^0 = 1$ vrchol, je pochopitelně jednoznačně určen;
- strom, který má 2^k vrcholů, vznikne ze dvou stromů s polovičním počtem, tedy 2^{k-1} vrcholy, které se sloučí tak, že kořen jednoho ze stromů se připojí jako nový syn ke kořeni druhého stromu.

Který z kořenů stromů se stane kořenem sloučeného stromu a který bude podřízen, není libovolné. Je vám jistě jasné, že pravidlo haldy (t.j. syn má větší klíč než otec) nás nutí jako nadřízený kořen (ten který zůstane kořenem nového stromu) volit ten z kořenů slučovaných stromů, který má menší klíč.

Zkuste si postupně vytvářet stromy s $1, 2, 4, 8, \ldots$ vrcholy pomocí knoflíku **Krok.** Na začátku je v levé polovině obrazovky zobrazen strom s jedním vrcholem, levá je prázdná. Pak vždy v lichém kroku se do pravé poloviny obrazovky vytvoří strom stejné velikosti a tvaru jako v levé polovině (jen s jinými klíči vrcholů) a v sudém kroku se oba stromy sloučí do levého okna na jeden strom dvojnásobné velikosti. Sledujte, jak pravidlo haldy určuje, který z kořenů slučovaných stromů se stane kořenem sloučeného stromu.

Knoflíkem **Zpět** se můžete vrátit ke stromu o řád menším (pokud strom má alespoň dva vrcholy). Jeden krok zpět odpovídá dvěma krokům dopředu.

Je vidět, že velikost stromů roste velmi prudce s jeho řádem, zatímco hloubka i stupeň kořene roste jen mírně.

Scéna: *Závislost tvaru binomické haldy na počtu vrcholů*

V této scéně budeme sledovat, jak tvar binomické haldy závisí na počtu jejích vrcholů. Kdykoli se zadá do číselného pole v ovladači přirozené číslo, vykreslí se na obrazovce halda s tímto počtem vrcholů.

Tvar haldy je dán dvěma pravidly: halda může obsahovat jen stromy, které byly popsány v předchozí scéně a nesmí obsahovat dva stromy stejného řádu neboli také stejné velikosti.

Uvážíme-li, že počet vrcholů stromu musí být mocninou dvojky (strom řádu k má 2^k vrcholů), pak existuje jednoduchý postup, jak určit, které

stromy použít v haldě o n vrcholech: Číslo n se napíše v binárním zápisu jako $b_{\ell-1}b_{\ell-2}\ldots b_1b_0$ (kde bit $b_{\ell-1}$ je nejvýznamnější a b_0 nejméně významný) a strom řádu k se použije právě tehdy, když bit b_k je roven 1 (a to se použije pochopitelně jeden strom řádu k), zatímco když b_k je 0, nebude v haldě žádný strom řádu k.

V takovém případě bude v haldě $\sum_{k=0}^{k-1} b_k 2^k$, co je přesně tolik, kolik je číslo s binárním zápisem $b_{\ell-1}b_{\ell-2}\ldots b_1b_0$, neboli n.

Pro ilustraci této metody je na pozadí na obrazovce, rozděleném do pásů, zapsán v binární soustavě počet vrcholů, tedy číslo, které se v dekadickém zápisu objevuje na ovladači a strom řádu k haldy se objevuje v pásu ve kterém je na pozadí zapsán k-tý bit binárního zápisu čísla n.

Scéna: *Slučování hald*

V této scéně začnu vysvětlovat první z operací, kterou binomická halda umožňuje efektivně provádět. Bude to operace, která se na první pohled zdá nejobtížnější - slučování dvou hald. Jak ale uvidíte, ostatní operace se velmi jednoduše vysvětlí za použití speciálních případů operace slučování, a proto je tento způsob výkladu nejvhodnější.

Na obrazovce jsou znázorněny dvě haldy s počty prvků zadanými na ovladači. Slučování si nejprve zkuste krokovat pro tyto hodnoty, jsou vybrány tak, že v průběhu slučování nastanou všechny důležité situace, ke kterým přitom může dojít.

Sloučení hald se provede tak, že dáme všechny stromy obou lesů, představujících zpracovávané haldy, dohromady. Tím vznikne nový les, který pochopitelně splňuje podmínku vertikální monotonie (stromy se neměnily) a všechny stromy mají přípustný tvar, ale může se stát, že halda obsahuje 2 různé stromy stejného řádu či velikosti. Abychom tuto potíž odstranili, provedeme následující:

pro $k = 0, 1, \ldots$ pokud strom obsahuje dva různé stromy řádu k, pak je sloučíme na jeden strom řádu $k + 1$.

Poznamenejme, že při provádění této operace se může stát, že v haldě nalezneme *tři* stromy řádu k: jeden z původní první haldy, druhý z původní druhé haldy a třetí, který vznikl během provádění operace ze stromů nižších řádů z obou hald. V tomto případě *libovolné* dva z nich sloučíme na strom vyššího řádu a třetí v haldě ponecháme beze změny. Postup použitý v animaci, kdy pro sloučení vybíráme stromy z původních hald, je pouze jedna z možností jak operaci provádět.

Na pozadí jsou na obrazovce zapsány binární zápisy velikostí původních hald, které se nacházejí v horní a střední vertikální části okna a v dolní vertikální části se postupně objevuje binární zápis počtu vrcholů sloučené haldy. Jistě vám neušlo, že slučování dvou hald velmi úzce imituje sčítání binárně zapsaných čísel, představujících počty vrcholů sčítaných hald. Sloučení dvou

hald přímo koresponduje s vytvořením přenosu do vyššího řádu při binárním sčítání.

Sloučení dvou stromů se provede v konstantním čase bez ohledu na tom, jak velké jsou slučované stromy. Je proto jasné, že řádově trvá operace sloučení stejně dlouho jako sečíst binární čísla představující počty vrcholů slučovaných hald. Operace je tedy mimořádně rychlá; počet sloučení je nejvýše roven délce binárního zápisu operandů, tedy $\lceil \log_2 n \rceil$, kde n je délka delšího ze sčítanců.

Scéna: *Přidávání do binární haldy*

Přidávání je velmi jednoduché. Vytvoříme si novou haldu, která obsahuje jediný vrchol (tedy je tvořena jediným stromem řádu 0) a přidávané číslo do tohoto vrcholu vložíme jako jeho klíč. Poté původní haldu a novou jednovrcholovou haldu sloučíme algoritmem vysvětleným v předchozí scéně.

Je zřejmé, že je nutné, aby uživatel zajistil, že do haldy je vždy přidáván takový prvek, který v ní ještě není. Na rozdíl od binárních vyhledávacích stromů a B-stromů není u hald jednoduché a rychlé zjistit, zda daný prvek v haldě již leží.

Zkuste si nyní postupné přidávání do binomické haldy, které je uspořádáno tak, aby připomínalo operaci slučování z předchozí scény (s tím, že po dokončení operace se výsledná halda ještě přesune z dolní polohy nahoru do horní třetiny okna).

Scéna: *Určování minima v binomické haldě*

Z pravidla vertikální monotonie vyplývá, že minimum v haldě se nachází v kořenu některého stromu haldy. Není ovšem možno předem říci, ve kterém stromě minimum je, proto je třeba prohlédnout všechny kořeny.

Operace je velmi rychlá, protože počet stromů v haldě je roven počtu jednotek v binárním zápisu čísla udávajícího počet vrcholů haldy a ten je pochopitelně nejvýše roven délce zápisu tohoto čísla, což je nejvýše $\lceil \log_2 n \rceil$.

Animace představuje procházení kořenů stromů (zelená barva vrcholu) a záznam dosavadního minimálního nalezeného klíče (červená barva).

Scéna: *Vynechávání z binomického stromu*

Než se dostaneme k vynechávání z obecné haldy, ukážeme si vynechávání z haldy, tvořené jediným stromem. V literatuře lze nalézt odlišné varianty, zde bude ukázána ta, který problém převede na vynechávání minima.

Zvolte si klepnutím myší libovolný vrchol, jeho klíč máme za úkol odstranit. Klíč se nejprve odstraní, ale vrchol kterému náležel, se ve stromu ponechá.

"Prázdno" ve vrcholu budeme považovat za klíč s hodnotou menší, než je hodnota jakéhokoli jiného klíče ve stromu. Tento "prázdný" klíč necháme stoupat vzhůru podobně jako tomu bylo v předchozí kapitole; zde prázdné místo pochopitelně vystoupá až do kořene stromu.

Když se bez klíče ocitne kořen stromu, tak jej odtrhneme a tím se strom, jak jistě vidíte na obrazovce, rozpadne na les, který obsahuje po jednom stromu každého řádu nižšího než byl řád původního stromu. Tento les tedy tvoří binomickou haldu, která je výsledkem operace.

Zkuste si operaci krokovat. Budete-li ji chtít provést opakovaně, musíte nejprve vytvořit nový strom.

Jak bude vypadat (a musí vypadat) výsledek operace je také možno určit jednoduchým výpočtem: původní strom má $n = 2^k$ vrcholů, kde k je řád stromu. Po odebrání jednoho vrcholu výsledná halda má $2^k - 1$ vrcholů a je známo, že platí

$$2^k - 1 = 1 + 2 + 4 + \cdots + 2^{k-1}.$$

Po odtržení jednoho vrcholu ze stromu proto musí vzniknout les, skládající se ze stromů všech nižších řádů.

Celá operace proběhne velmi rychle. Jediné, co je na ní zdlouhavé je probublávání prázdného klíče vzhůru; počet kroků, který je k tomu třeba, je nejvýše úměrný hloubce stromu, což je $k = \log_2 n$.

Scéna: *Vynechávání z binomické haldy*

Poslední operací, kterou ještě schází vysvětlit, je vynechávání z obecné haldy. S přihlédnutím k už známým operacím je to ale jednoduché.

Strom, které obsahuje vrchol, jehož klíč se má vynechat, se vyjme z haldy a vytvoří se z něho druhá halda, obsahující jediný strom. Z této druhé haldy se klíč vynechá tak, jak to bylo vysvětleno v minulé scéně. Nakonec se obě dvě haldy známým způsobem sloučí.

Zkuste si (třeba i opakovaně) zvolit vrchol a pak krokovat vynechávání jeho klíče.

Kapitola 6

Faktorová množina

V některých případech je třeba uchovávat informaci o rozkladu jisté pevné množiny M do navzájem disjunktních tříd, které pokrývají celou množinu. Konkrétním případem je Kruskalův algoritmus pro vyhledávání minimální kostry grafu, který je uveden v části o grafových algoritmech.

Budeme vyžadovat, aby bylo možno provádět tři operace:

- **Inicializace:** nastaví datovou strukturu tak, že popisuje rozdělení množiny M do jednoprvkových tříd;
- **Určení třídy:** je dán prvek množiny M a je třeba určit (jednoznačně danou) třídu, ve které tento prvek leží;
- **Sloučení tříd:** dvě dané různé třídy rozkladu se sloučí v jednu.

Datová struktura, umožňující provádět tyto operace se obvykle nazývá *faktorová množina*.

Rozklad je po provedení inicializace diskrétní, každý prvek množiny M představuje jednu třídu. Má-li tedy množina M celkově n vrcholů, je na začátku v rozkladu n tříd. Každé provedení operace sloučení tříd sníží počet tříd o 1, takže pokud se inicializace provede pouze na začátku používání struktury, může se celkově sloučení tříd provést nejvýše $(n-1)$-krát. Počet provedení operace určení třídy je ovšem zcela libovolný.

U Kruskalova algoritmu se operace určení třídy používá tak, že je dána hrana grafu s množinou vrcholů M a má se určit, zda její konce leží ve stejné třídě. Otázka se zodpoví tak, že se operace provede na oba konce hrany a výsledky se porovnají. Pokud leží konce hrany v různých třídách, tyto třídy se pak sloučí.

Scéna: *Rozklad obarvením*

V této scéně ukážeme jednu implementaci faktorové množiny. Každý prvek je označen jistým údajem, označujícím třídu, a to tak, že dva prvky mají tyto údaje stejné právě tehdy, když patří do stejné třídy. Údaj může být v počítačovém programu číslo, na obrazovce v této scéně používáme barvu, kterou je prvek obarven.

Inicializace přiřadí prvku výhodně jeho jméno nebo identifikační označení. Na obrazovce označíme prvky navzájem různými barvami. Určení třídy prvku se redukuje na předání údaje spojeného s prvkem.

Sloučení dvou tříd se provede tak, že se jedna třída vybere a údajem, kterým jsou označeny všechny její prvky se označí i prvky druhé třídy.

Inicializace pochopitelně vyžaduje provést s každým prvkem jednoduchou operaci (označení) a proto vyžaduje čas úměrný velikosti množiny. Jelikož se inicializace většinou provádí pouze jednou, nezpůsobuje to žádný problém.

Určení třídy obsahující prvek je okamžité, zato operace sloučení tříd může být časově náročná, neboť doba k jejímu provedení je úměrná velikosti třídy, u jejíchž prvků přepisujeme označení.

Pro rychlost slučování tříd je výhodné aby třída, jejíž označení přepisujeme, byla menší z obou slučovaných tříd (pokud jsou stejně velké, pak je jedno, kterou pro přepisování vybereme). V takovém případě je třeba, aby u každé třídy byl znám i počet jejích prvků, aby nemusel být pokaždé pracně určován. To je jednoduché zajistit, po inicializaci má každá třída 1 prvek a při sloučení je počet prvků sloučené třídy roven součtu prvků výchozích tříd.

Množinu M si můžeme na obrazovce vytvořit náhodně volbou **Inicializace** s tím, že bude každý prvek mít jinou barvu a její velikost je dána číslem vedle návěští N = na ovladači. Klepnutí na prázdné místo vytvoří nový prvek, který bude patřit jako jediný do nové třídy. Klepnutí na existující prvek tento prvek vynechá. Je třeba uvést, že tyto činnosti nejsou standardními operacemi faktorové množiny a slouží pouze pro přípravu rozkládané množiny M, pro jinou volbu operace než je Inicializace pracuje myš jiným způsobem.

Při volbě **Vyhledání** se po dotazu klepnutím na některý prvek levým knoflíkem po straně objeví odpověď: barva třídy obsahující zvolený prvek, což je současně i barva tohoto prvku.

Sloučení se provede při volbě **Slučování** tak, že se nejprve poklepne na dva vrcholy patřící do dvou různých tříd. Barvy tříd se objeví po straně a objeví se také knoflíky pro přebarvování. Knoflík **Krok** přebarvuje po jednom prvku, knoflík **Sloučení** přebarví třídu najednou, knoflíky **Zpět** a **Začátek** vrací výpočet zpět po jednom prvku nebo úplně. Prvky slučovaných tříd jsou označeny bílým podkladem. Knoflíky **Dokonči** a **Zruš** prováděnou operaci zcela dokončí nebo zruší a je možno zvolit jinou dvojici tříd pro slučování.

Na ovladači je také možno vybrat způsob, jak určit, která ze slučovaných komponent si ponechá svoji barvu. Při volbě **První** je to komponenta, kterou

uživatel vybral první, při volbě **Náhodná** se jedna z nich vybere náhodně, při volbě **Větší** si barvu *ponechá* větší z nich (při stejné velikosti systém vybere náhodně) a nakonec je jako odstrašující případ dána i možnost **Menší,** kdy si menší komponenta ponechá barvu a větší se přebarvuje.

Operace vyhledávání je nezajímavá, ale doporučuji si operacemi slučování postupně spojit všechny třídy do jedné s krokováním přebarvování po prvcích a alespoň pro volby **Náhodná** a **Větší.**

Scéna: *Rozklad obarvením - nejhorší případ*

U rozkladu obarvením je určení třídy okamžité. V této scéně ale uvidíme, že slučování tříd může může celkově vyžadovat dosti velký počet kroků (tedy přebarvení jednotlivých vrcholů). Pro jednoduchost je v této scéně počet vrcholů, označený n, roven mocnině 2.

Algovize si třídy pro slučování vybírá sama způsobem, který vede u tohoto algoritmu na největší počet provedených kroků. Nejprve se všech n jednoprvkových tříd sloučí do $n/2$ dvojprvkových, pak se dvojprvkové třídy sloučí do $n/4$ čtyřprvkových atd. Jelikož vždy slučujeme třídy stejné velikosti, všechny volby **První, Náhodná, Větší** i **Menší** dělají totéž a proto volba varianty na ovladači není použita. Je zřejmé, že fází, v rámci kterých zpracováváme třídy téže velikosti, je $\log_2 n$ a v každé fázi se přebarví přesně polovina prvků. Celkově tedy v rámci slučování tříd provedeme $0,5n \log_2 n$ elementární přebarvení vrcholů.

Scéna: *Rozklad popsaný stromem*

Předchozí způsob rozkladu není možné příliš vylepšit. V této scéně ukážeme jiný způsob popisu rozkladu, ze kterého můžeme vytvořit velmi výhodnou implementaci faktorové množiny.

Každá třída rozkladu je vnitřně reprezentována jako strom, kde každý prvek třídy má ukazatel, který ukazuje na jeho předchůdce ve stromu. Jdeme-li podle ukazatelů dostaneme se nakonec do kořene stromu, který představuje reprezentanta třídy. Pro kořen v různých implementacích buď ukazatel není definován nebo (jako zde) ukazuje sám na sebe. Podle ukazatele lze tedy reprezentanta třídy snadno odlišit od jemu podřízených prvků.

V závislosti na volbě na ovladači jsou ukazatele zobrazeny jedním ze tří možných způsobů:

- buď jsou stále zobrazeny všichni ukazatelé;
- nebo jsou ukazatelé skryti a pro zobrazení ukazatele zvoleného prvku je třeba na prvek klepnout;
- nebo jsou ukazatelé skryti a klepnutím na prvek se zobrazí celá cesta ze zvoleného prvku až k reprezentantu třídy, ve které leží.

Inicializace namíří ukazatel každého prvku na sebe, což znamená, že je reprezentantem sebe sama.

Dotaz na příslušnost třídy se vyhodnotí tak, že se po cestě z ukazatelů postupuje z dotazovaného prvku až do reprezentanta třídy. Na rozdíl od předchozí scény je tedy složitější. Postup po cestě je v appletu znázorněn pohybem barevného kolečka.

Sloučení tříd se provede tak, že ukazatel reprezentanta jedné třídy se přesměruje na reprezentanta druhé třídy. Změněný ukazatel je do provedení další operace barevně zvýrazněn. Na rozdíl od předchozí scény je tedy slučování velmi rychlé. Jestliže ovšem zadáme třídy určené ke sloučení nikoli jejich reprezentanty, ale jejich obecnými prvky, operace se prodlouží o určení reprezentantů tříd.

Při sloučení se prodlouží o 1 krok cesta k reprezentantovi pro prvky třídy, jejíž reprezentant byl připojen jako podřízený ke druhé třídě. Je proto výhodné aby připojovaná třída měla menší počet prvků. Podobně jako v úvodní scéně rozkladu obarvením je několik volitelných možností, která komponenta se bude připojovat ke které.

Vyzkoušejte si operace s touto implementací faktorové třídy; obsluha scény je velmi podobná jako ve scéně předchozí. Navíc přibyla volba způsobu zobrazování ukazatelů.

Scéna: *Rozklad stromem - nejhorší případ*

Velikost množiny je opět mocnina 2 a příkazy ke slučování tříd dává Algovize stejné jako ve scéně o nejhorším případu rozkladu obarvením. Zde ale proběhne dvou sloučení tříd v konstantním čase, takže celkově je počet kroků strávených slučováním úměrný $n + n/2 + n/4 + \cdots + 1 = 2n - 1$. Na konci k-té fáze, kdy třídy rozkladu mají 2^k prvků, jsou tvořeny stromy hloubky k, ve kterých na kořen ukazuje k jeho následníků. Je zajímavé že stromy v této scéně vypadají stejně jako stromy, které jsme používali v předchozí kapitole o binomické haldě.

Nevýhodou rozkladu stromem ale je, že určení reprezentanta třídy může trvat déle. V nejhorším případě v koncové třídě s n prvky se nachází jeden prvek, pro který je při určení třídy nutné přejít $\log_2 n$ ukazatelů a lze ukázat, že i střední délka cesty ke kořeni (reprezentantu třídy) je této hodnotě blízká.

Ani v této scéně postup nezávisí na variantě, která třída bude připojována ke které. Lze ukázat, že pro variantu, ve které se menší třída připojuje k větší, je příklad uvedený v této scéně tím nejhorším, co může nastat.

Mohlo by se zdát, že v případě, kdy se v průběhu používání datové struktury provede velké množství dotazů na třídu (jejich počet na rozdíl od sloučení tříd není omezen), je rozklad obarvením výhodnější. V následující scéně ale ukážeme, že tuto nevýhodu rozkladu stromem lze snadno odstranit.

Scéna: *Zkracování cesty*

V této scéně se ukáže jednoduché vylepšení způsobu se stromem ukazatelů, které výrazně sníží čas strávený určováním třídy, obzvláště pokud tato operace se provádí velmi často.

Jestliže se provede dotaz na příslušnost do třídy, projde se celá cesta od dotazovaného prvku až do reprezentanta třídy. Jako vedlejší výsledek operace se zjistí pro všechny prvky prošlé cesty, kdo je reprezentantem třídy, do které náleží. Proto je vhodné po provedení operace *přepojit* ukazatele všech těchto prvků na nalezeného reprezentanta. Při příštím dotazu na příslušnost kteréhokoli z těchto prvků se již dotaz zodpoví v konstantním čase přechodem jediného ukazatele přímo k reprezentantovi. Jestliže se tedy po nějakou dobu neprovádí slučování, ale dotazy na příslušnost, poklesne doba potřebná na provedení těchto operací až na konstantní čas potřebný k přechodu po jediném ukazateli bez ohledu na velikost rozkládané množiny.

Část II

Třídění

Všechny kapitoly této části se věnují třídění posloupnosti čísel. V prvních čtyřech kapitolách je naším cílem je setřídit posloupnost čísel $a_1, \ldots, a_n$, to znamená pozměnit jejich pořadí tak, že vzniklá posloupnost je buď neklesající nebo nerostoucí. Elementárními operacemi, které přitom používáme, je porovnání dvou čísel podle velikosti a prohození dvou čísel, které budeme provádět na různé dvojice tak dlouho, dokud není posloupnost setříděná. Zajímat nás bude především, kolik elementárních porovnání a prohození je pro setřídění potřeba provést.

Druh čísel, které tvoří posloupnost nebude důležitý; algoritmům zde uvedeným stačí, aby bylo o dvou vybraných číslech možno rozhodnout, které z nich je větší (a nebo že jsou si rovna). Čísla tedy mohou být celá čísla, reálná čísla (typy float, double, a pod.), ale ne komplexní čísla, protože pro ty uspořádání není definováno.

V appletech prvních čtyř kapitol budou čísla obvykle reprezentována graficky jako svislé sloupečky jisté výšky, která je úměrná velikosti čísla, stojící na vodorovné základně (s částečnou výjimkou pro bubblesort, kde je možno je zobrazovat také jako vodorovné sloupce opřené zleva o svislou základní linii). Applety budou používat pouze nezáporná čísla, aby sloupce neležely pod základnou, což je ale nepodstatné omezení, vzhledem k tomu, že nás nezajímají absolutní hodnoty čísel, ale jen jejich vzájemné vztahy.

Ve všech případech bude možné nastavit délku tříděné posloupnosti použitím pole vedle **N=** na ovladači a nová náhodná vstupní posloupnost se může vytvořit knoflíkem **Nová posloupnost.** Hodnota libovolného čísla v posloupnosti se dá změnit tažením horní části odpovídajícího sloupce nahoru či dolů (tato možnost je ale jen před započetím výpočtu a je blokována během jeho krokování nebo animace).

V předposlední kapitole se budeme zabývat slabší úlohou než setřídění: určení k-tého největšího prvku (čímž se míní prvek, který se v posloupnosti objeví na k-tém místě po jejím setřídění). Nejčastěji se setkáváme s její speciální formou - určení mediánu. Mediánem se rozumí $\lfloor n/2 \rfloor$-tý prvek posloupnosti. Pochopitelně je možné úlohu vyřešit setříděním posloupnosti a odpočítáním k-tého prvků, ale zde ukážeme postupy, které k cíli vedou podstatně rychleji.

Poslední kapitola se také zabývá tříděním pomocí porovnávání a prohazování prvků tříděné posloupnosti, ale používáme jiný model: třídící obvod. Tento model zahrnuje paralelismus: porovnání a záměna disjunktních dvojic může být prováděna současně v různých místech obvodu, což velmi zkrátí dobu pro setřídění.

Kapitola 7

Mergesort

Mergesort je jedním z nejstarších, ale také nejrychlejších třídicích algoritmů, navrhl jej John von Neumann v roce 1945. Jeho nevýhodou ve srovnání s ostatními zde popisovanými algoritmy je především potřeba velkého množství pomocné paměti.

Scéna: *Slučování*

Podstata Mergesortu je prostá. Tříděná posloupnost $a_1, \ldots, a_n$ se rozdělí na dvě přibližně stejné části, například na $a_1, \ldots, a_m$ a $a_{m+1}, \ldots, a_n$, kde $m = \lfloor n/2 \rfloor$ a pak se setříděné posloupnosti sloučí.

Sloučení dvou setříděných posloupností se provádí tak, že se opakovaně odebírá ten z prvních prvků obou posloupností, který je menší, dokud se jedna posloupnost nevyprázdní. Zbytek druhé posloupnosti se pak přidá na konec slučované posloupnosti (najednou nebo po prvcích dle varianty algoritmu).

Tato scéna ukazuje po krocích nebo v plynulé animaci sloučení dvou setříděných posloupností. Slučování je tak jednoduchou a přirozenou operací, že jistě není třeba ji podrobněji objasňovat.

Knoflíkem **Nové posloupnosti** je možno vytvořit nové vstupní posloupnosti a zkusit si slučování znovu, lze nastavit i délky slučovaných posloupností, které nemusí být stejné.

Scéna: *Iterativní Mergesort*

Na třídící algoritmus Mergesort je možno se dívat tak, že nejprve se na posloupnost díváme jako na n setříděných jednočlenných posloupností. Pak jednoprvkové posloupnosti spárujeme a každý pár sloučíme do jedné sloučené posloupnosti dvojnásobné délky. Sloučíme tedy první prvek s druhým, třetí s čtvrtým atd. Když je posloupnost tvořena setříděnými dvojčlennými posloupnostmi, opět je spárujeme a slučujeme do setříděných čtveřic, čtveřice

spárujeme a slučujeme do setříděních osmic atd. až nakonec dostaneme setříděnou posloupnost. V této scéně je délka posloupnosti mocninou dvojky, takže ji lze stále perfektně půlit.

Obrazovka ukazuje systém rámečků, do kterých se slučované posloupnosti přepisují. Do prázdného rámečku se sloučí posloupnosti z rámečků bezprostředně pod ním. Konkrétní implementace Mergesortu bude vypadat trochu jinak, ale grafická struktura použitá v této scéně ukazuje pěkně logické schéma algoritmu.

Pořadí slučování je v zásadě lhostejné: jakmile jsou dva rámečky, umístěné pod společným nadřazeným rámečkem, obsazeny setříděnými posloupnostmi, lze začít je slučovat. Zkuste si výpočet pro různé možnosti volby na ovladači, které odpovídají různým implementacím. Je také možno provádět dvě slučování do dvou různých rámečků současně, pokud by k dispozici bylo paralelní výpočetní zařízení, které to umožňuje. Na ovladači lze volit i několik základních paralelních postupů slučování. Doporučuji si je všechny vyzkoušet. Uvidíte ale, že na konkrétní volbě příliš nezáleží a je dána spíše osobními preferencemi programátora.

Scéna: *Rekurzivní Mergesort*

Zatímco v předchozí scéně bylo slučování plánováno zdola nahoru, nyní bude plánováno shora dolů. Tento způsob je vhodný, pokud jsou k dispozici rekurzivní procedury, pomocí nichž pak lze algoritmus naprogramovat velmi snadno. Jak ale uvidíte, prováděná porovnávání prvků a jejich přesuny budou obdobné jako při iterativním plánování výpočetního procesu.

Na začátku výpočtu se horní rámeček červeně zvýrazní a objeví se v něm otazník. Označuje to, že úkolem je vytvořit setříděnou posloupnost zahrnující všechny prvky.

Algoritmus pracuje tak, že dokud není posloupnost setříděna, což bude dokud je na obrázku alespoň jeden rámeček s otazníkem, pak pro některý rámeček R s otazníkem, který má bezprostředně pod sebou alespoň jeden rámeček bez otazníku, se provede jedna z následujících činností:

- je-li bezprostředně pod rámečkem R jeden nebo oba prázdné rámečky, pak se do jednoho z těchto prázdných rámečků umístí otazník (požádáme program, aby do něho dostal setříděnou posloupnost);
- jsou-li v obou rámečcích bezprostředně pod rámečkem R již setříděné posloupnosti, pak se tyto posloupnosti sloučí do rámečku R;

Poznamenávám, že jsou-li pod rámečkem R má šířku dva jednoprvkové rámečky, pak jelikož jednoprvková posloupnost je setříděná, postupuje se podle posledního bodu.

Provádění rekurzivního schématu je možno na ovladači upřesnit tak, že pokud oba rámečky pod nejnižším rámečkem R s otazníkem jsou zatím prázdné,

pak se otazník dá nejprve do levého, pravého, respektive náhodně vybraného z nich. Implementována je také možnost paralelního výpočtu, kdy se oba podřízené rámečky zpracovávají současně.

Scéna: *Posloupnost obecné délky*

Pokud délka posloupnosti není mocnina 2, pak alespoň některé z rámečků není možno rozdělit na dva podřízené rámečky *stejné* šířky. To vede k tomu, že proces dělení rámečků na přibližné poloviny se někdy zastaví dříve (v menší hloubce) a jindy později (ve větší hloubce). Pro dosažení uniformnosti schématu i procesu zde byly do struktury rámečků zařazeny i do druhé vrstvy odspodu rámečky šířky 1 (mají červenofialovou barvu) a do nich se z (jediného) rámečku pod ním neslučuje, ale pouze přesouvá jediný prvek. Největší počet těchto formálně přidaných červenofialových rámečků je pokud je n ve tvaru $1 + 2^k$, kde k je přirozené číslo; pak jich je $n - 2$ (v nejnižší vrstvě se pouze sloučí dva prvky a zbývajících $2^k - 1$ prvků se přenáší - zkuste si například $n = 33$).

Zkoušejte si různé délky tříděné posloupnosti a sledujte možné tvary struktury rámečků. Algoritmicky tato scéna nepřináší nic nového.

Poznamenávám, že sice je nejvýhodnější pro rychlost výpočtu dělit rámeček na dva, jejichž šířka se rovná nebo liší o 1, ale méně vyvážené dělení, pokud by zaručovalo, že šířky rámečků vzniklých rozdělením se nebudou příliš lišit, je také použitelné a dává i dobrou rychlost třídění. V této knize však neoptimální dělení rámečků rozebírat nebudeme.

Scéna: *Překrytí*

Struktura rámečků, použitá v předchozích scénách sice dává jasný pohled na logickou strukturu algoritmu, ale pro praktické použití by byla velkým plýtváním pamětí.

Rozpomeňte se na předchozí scény nebo se k nim vraťte. Pro libovolnou svislou přímku platí, že protíná nejvýše dva neprázdné rámečky. Pokud jsou takové rámečky dva, pak ještě navíc leží v sousedních řádcích, tedy jeden v lichém řádku a jeden v sudém řádku.

Vycházejíce z toho, používáme v této scéně jen dva řádky rámečků a do jednoho (s překrytím) vložíme liché řádky struktury z předchozích scén a do druhého sudé řádky. Opravdu však nakreslíme jen ty rámečky, které nejsou prázdné, tedy obsahují alespoň jeden prvek tříděné posloupnosti a proto se v překrytém nakreslení struktury *nepřekrývají*.

Pro naši překrytou strukturu tedy platí, že neprázdné rámečky se tedy nikdy nepřekrývají. Zkuste si výpočet v libovolné z výše popsaných variant a pro libovolnou délku posloupnosti znovu a sledujte, jak se v předchozích scénách popsané postupy projevují v překryté struktuře.

Pokud vám to pomůže, můžete v libovolném okamžiku vypnout volbu **Překrytí** a struktura rámečků se znovu objeví v rozvinuté podobě.

Překrývaný způsob je paměťově výhodná implementace Mergesortu. Jsou sice možná ještě další drobná vylepšení, ale ty již není účelné zde ukazovat.

V závislosti na délce posloupnosti někdy skončí setříděná posloupnost ve stejném řádku překryté struktury jako byla na začátku uložena výchozí posloupnost a jindy v druhém z nich (záleží to na tom, zda parita horního a dolního řádku ve struktuře podle předchozích scén je stejná nebo odlišná). Pokud je to požadováno, pak v druhém případě se postup může doplnit překopírováním setříděné posloupnosti do výchozí oblasti. Tato úprava znázorněna není.

Kapitola 8

Quicksort

Quicksort je způsob třídění, který byl navrhl v roce 1962 C. A. R. Hoare. Je-li použit v jednoduché podobě, je sice možno nalézt posloupnosti, které třídí podstatně pomaleji než například mergesort nebo heapsort, ale své jméno si zaslouží, protože při třídění posloupností "ze života" patří mezi absolutní špičku a podle některých autorů je dokonce vůbec nejrychlejší ze základních algoritmů pro třídění. Některé jeho varianty pak odstraňují nevýhodu možnosti pomalého výpočtu pro některé vstupy.

Scéna: *Myšlenka quicksortu*

V této scéně bude graficky ilustrována myšlenka, na které je založen, popis vhodný pro implementaci bude ukázán v následující scéně.

Čísla, která je třeba setřídit, jsou znázorněna sloupečky, jejichž výška odpovídá velikosti čísla. Klepněte na knoflík **Krok.** Algoritmus zvolí číslo, které se nazývá *pivot*. Jeho velikost je dána výškou červeného trojúhelníka, který se objeví na levé straně rámečku, ve kterém jsou nakresleny sloupce pro setřídění. Pivota nelze volit zcela libovolně, volba bude podrobněji rozebrána dále.

Po následujícím klepnutí na knoflík **Krok** se v rámečku zleva doprava nakreslí vodorovná červená čára, leží ve výšce odpovídající pivotu. Některé sloupečky červená čára propíchne a ty ztmavnou; sloupečky nižší než je hodnota pivota barvu nemění. Za "napíchnuté" se považují i sloupce, kterých se červená čára jen dotkne.

Další krok (knoflík **Krok)** pivotovou čáru zvedne nahoru. Spolu s ní se vyzvednou nahoru "napíchnuté" tmavší sloupečky, které odpovídají hodnotám větším nebo rovným pivotu. "Nenapíchnuté" světlejší sloupečky se nepohnou.

V dalším kroku se napíchnuté tmavé sloupečky, visící ve výšce, shrnou doprava a nenapíchnuté světlé sloupečky, ležící na základové čáře, se shrnou

doleva. Následuje krok, ve kterém se tmavé sloupce spustí dolů na základovou čáru a obnoví se jejich původní barva.

Výsledkem je, že se sloupečky nižší než pivot shromáždí nalevo a sloupečky větší nebo rovné pivotu se shromáždí napravo. V této scéně jsou sloupečky v každé z obou skupin v původním pořadí, ale toto není pro činnost algoritmu nutné a některé další implementace pořadí ve skupinách prohází.

V dalším kroku se původní rámeček, objímající všechny sloupečky, nahradí dvěma rámečky, levý pro nenapíchnuté sloupečky a pravý pro sloupečky, které byly napíchnuty.

Je zřejmé, že libovolný sloupec v levém rámečku je menší než libovolný sloupec v pravém rámečku. Abychom tento fakt zdůraznili, jsou rámečky stejně vysoké, jako nejvyšší sloupec, který v nich je obsažen; navíc je spodní část pozadí rámečku tmavší až do výšky nejnižšího sloupce v rámečku. Vidíte, že opravdu výška levého rámečku je menší než výška spodní tmavé části v pravém rámečku.

Nyní stačí uspořádat nejprve levý rámeček a pak pravý rámeček a nechat je tak jak jsou položeny vedle sebe a posloupnost bude setříděna. Pochopitelně také lze setřídit nejprve pravý rámeček a pak levý a nebo, pokud lze výpočty provádět paralelně, je možné je setřídit najednou. Uděláme to opakováním stejného postupu: pro setřídění rámečku si zvolíme nového pivota, podle něj sloupečky z rámečku rozdělíme do menší skupiny vlevo a větší skupiny vpravo, které zase třídíme stejným postupem, atd. Klepáním na knoflík **Krok** si projděte celý výpočet.

Pokud ovšem v rámečku mají všechny sloupečky stejnou výšku, speciálně pokud je v něm jediný sloupeček, pak ovšem je rámeček setříděný a žádnou činnost (volba pivota, separace sloupečků) v něm neprovádíme.

Při výpočtu je vždy jeden rámeček zpracováván a řada dalších na zpracování čeká, nebo již zpracována (všechny sloupečky v rámečku stejně vysoké). Zpracovávaný rámeček je barevný, rámečky čekající na zpracování jsou šedivé. Rámeček s konstantní (např. jednoprvkovou) posloupností už nekreslíme jako rámeček, pouze se vykreslí sloupečky v něm obsažené tmavě zelenou barvou.

Nyní se vrátíme k volbě pivota: aby se postup nezacyklil, je třeba, abychom při separaci sloupečků v rámečku dostali dvě neprázdné skupiny. Tím bude dáno, že obě skupiny budou menší než původní množina sloupečků rámečku. Je tedy třeba pivota volit tak, aby byl *větší* než nejmenší prvek v rámečku (tedy levá skupina bude neprázdná), ale *menší nebo roven* největšímu prvku v rámečku (pak bude i pravá skupina neprázdná). Jelikož volbu pivota provádíme jen v případě, že posloupnost v rámečku není konstantní (její minimum je ostře menší než maximum), je taková volba vždy možná, například jako maximum prvků.

V implementaci na obrazovce jsou výšky sloupečků celá čísla, proto je pivot volen jako celé číslo z intervalu $[min + 1, max]$, kde min a max jsou

výšky nejnižšího, respektive nejvyššího sloupečku. Jelikož ho volíme pokud $min < max$, je tento interval neprázdný; v této scéně je volen náhodně se stejnou pravděpodobností mezi všemi celými čísly v tomto rozmezí. Určení minima a maxima v počátečním rámečku se provede například postupným probráním všech sloupečků v rámečku (tato činnost není vizualizována), pro další rámečky se minima a maxima mohou určit současně s porovnáváním pivota s prvky většího rámečku, ze kterého rozštěpením vznikly.

Scéna: *Implementace výměnami*

V této scéně ukážeme, jak se separace sloupečků uvnitř rámečku provádí velmi výhodným způsobem. Volba pivota se opět provádí náhodně v rozmezí uvedeném v předchozí scéně, ale "napichování" sloupečků není animováno, pivotová červená čára se nakreslí okamžitě celá a vyšší sloupečky také ztmavnou mžikově.

Je-li pivot zvolen, nastaví se v rámečku dva ukazatele sloupečků. Jeden, znázorněný modrým trojúhelníkem pod základovou čárou, je na začátku nastaven na levý sloupeček rámečku, druhý, zelený, začíná pod pravým sloupečkem rámečku. Levá skupina sloupečků nižších než pivot se bude shromažďovat vlevo od modrého ukazatele, pravá skupina sloupečků vysokých alespoň jako pivot se bude objevovat napravo od zeleného ukazatele.

V první fázi se modrým ukazatelem pohybuje vpravo tak dlouho, dokud nenarazí na napíchnutý tmavší sloupec, který představuje hodnotu větší nebo rovnou pivotu, tedy na hodnotu, která nepatří do levé skupiny. Může se stát, že se ukazatel vůbec nepohne, pokud na takovou hodnotu ukazoval již na začátku.

V druhé fázi se zase zelený ukazatel pohybuje vlevo tak dlouho, dokud nenarazí na nenapíchnutou světlejší hodnotu, menší než pivot.

Po skončení obou fází se sloupce, na které ukazují ukazatele, prohodí. Tím se k oběma ukazatelům dostanou hodnoty, které patří do jejich skupin a jejich postup může pokračovat.

Po záměně sloupců se opakuje znovu první a druhá fáze a prohození a to tak dlouho, dokud si modrý a zelený ukazatel neprohodí místa; modrý bude o jednu polohu vpravo od zeleného. Pak se již prohození neprovede a sloupce jsou správně separovány.

Tento způsob separace sice nezachovává pořadí v napíchnuté a v nenapíchnuté skupině, to však nevadí, jak již bylo uvedeno v minulé scéně.

Velikou výhodou celého postupu je, že nepotřebuje prakticky žádnou pomocnou paměť, pouze dva ukazatele a případně jedno pomocné místo na prohození sloupečků.

Scéna: *Volba pivota*

Ukazuje se, že rychlost quicksortu je velmi silně závislá na způsobu, jak je volen pivot. Tato scéna opakuje scénu předchozí, ale uživatel může volit mezi několika velmi odlišnými způsoby volby pivota a sledovat, jak se projevují na rychlosti výpočtu.

Pro porovnání voleb Algovize v pravém horním rohu obrazovky ukazuje počet porovnání a počet prohození čísel, které algoritmus provedl. Zkuste si pro tutéž vstupní posloupnost porovnat tyto počty pro jednotlivé varianty. Náhodnou volbu zkuste opakovaně, protože v tomto případě může pokaždé počet operací vyjít jinak, i když je vstup stejný, protože průběh výpočtu závisí na náhodě.

Při volbě **Nejlepší** je pivot volen tak, aby si byly skupiny vzniklé separací sloupců v rámečku velikostně co nejpodobnější - stejně velké, pokud je sloupců sudý počet a lišící se o jeden prvek, je-li sloupců lichý počet. Uvidíte, že algoritmus postupuje velmi rychle, protože je to nejlepší možná strategie. Má jedinou nevýhodu: určení pivota s uvedenou vlastností by si vyžádala dosti složitý výpočet, který by celkovou dobu třídění značně prodloužil. V naší animaci to není poznat, protože volba pivota není animována ani započítávána do počtu operací, ale ve skutečnosti má postup s optimálními volbami pivota jen teoretický význam.

Při volbě **Nejhorší** je volba pivota záměrně prováděna způsobem, který je co nejméně výhodný. Je volen tak, aby velikost skupin, na které je daný rámeček separován, byla co nejvíce nevyvážená. Dosáhneme toho tak, že jako pivota volíme buď výšku maximálního prvku v rámečku, nebo naopak minimum +1. Algoritmus v této scéně vybírá z uvedených dvou možností tu horší (tedy menší velikost menší skupiny). Pokud je posloupnost prostá (neobsahuje dvě stejná čísla), pak jedna ze skupin bude obsahovat jediný sloupeček, druhá všechny ostatní.

Zkuste si výpočet a poznáte, že opravdu trvá citelně déle než pro nejlepší volbu pivota.

Při volbě **Průměrový** se pivot volí jako $\lceil (min + max)/2 \rceil$. Pokud jsou prvky mezi minimem a maximem rozloženy rovnoměrně, lze očekávat kvalitní volbu pivota přinášející podobně veliké rozdělené skupiny, ale je snadné vymyslet příklad, kdy budou skupiny rozděleny velmi špatně. Pro různé vstupy si zkuste tento způsob porovnat s ostatními. Dá se ukázat, že pro náhodné posloupnosti se metoda chová dobře, může mít ale potíž s některými záměrně sestrojenými vstupy.

Při volbě **Prostřední** se jako pivot volí hodnota prvku, který se nachází v polovině posloupnosti. V praxi mívají často posloupnosti již částečně monotónní charakter a pak je tato volba dosti podobná volbě průměrové. Pro náhodné posloupnosti se také chová dosti podobně jako náhodná volba uvedená v následujícím odstavci. Není ale těžké vymyslet jako schválnost posloup-

nost, pro kterou tato volba dosáhne nejhoršího možného počtu porovnání a prohození.

Při volbě **Náhodný** se pivot volí náhodně v rozmezí $[min + 1, max]$. Dá se ukázat, že v takovém případě algoritmus pracuje s velmi velkou pravděpodobností velmi rychle, provede jen o málo porovnání a prohození čísel více než v případě optimální volby, ale určení pivota je rychlé a nedochází k neúměrnému prodloužení doby výpočtu jako při první volbě. Porovnejte počet operací s nejlepší (ale zdlouhavou) volbou pivota. Quicksort s náhodnou volbou pivota je jednou z nejlepších metod třídění pro praktické potřeby.

Scéna: *Pořadí zpracování rámečků*

V této scéně je třídění quicksortem zopakováno ještě jednou. V okamžiku, kdy je skončeno zpracování jednoho rámečku se další z čekajících rámečků vybere pro zpracování náhodně a ne jako nejlevější čekající rámeček, jak tomu bylo v předchozích scénách. Uvidíte, že tato libovůle nemá žádný vliv na správnost výpočtu ani na výpočetní dobu. Programátor proto může volit pořadí zpracování rámečků podle potřeby tak, jak je pro něho nejvýhodnější.

Scéna: *Paralelní výpočet*

Pokud by byl pro výpočet k dispozici paralelní počítač, třídění by mohlo probíhat velmi rychle. Po separaci rámečku do dvou skupin by se skupiny mohly třídit současně různými procesory nebo skupinami procesorů. Paralelní implementace je ukázána v této scéně. V ovladači je přitom možno nastavit některou z výše uvedených způsobů volby pivota: nejlepší, nejhorší, průměrovou, prostřední a náhodnou.

Scéna: *Quicksort*

Poslední scéna nic nového nepřináší, ale je možno si s algoritmem pohrát a zvolit libovolné možnosti uvedené výše. Separaci prvků v rámečku lze provádět s "propichováním" nebo s výměnami. Pivota lze volit stejně jako bylo uvedeno výše. Rámeček pro zpracování může být volen jako nejlevější, nejpravější nebo náhodný. Změna nastavení pochopitelně resetuje výpočet a obnoví původní posloupnost. Je též možno požádat o zadání nové posloupnosti stejné nebo změněné délky. Pro vzájemné porovnání variant se v rohu obrazovky ukazuje počet porovnání čísel, provedených během výpočtu.

Kapitola 9

Heapsort

Tato kapitola úzce navazuje na kapitolu o haldě v části o datových strukturách. Pokud jste si tuto kapitolu ještě neprostudovali, učiňte tak nyní a pak se teprve vraťte sem.

Heapsort je algoritmus, který popsal J. W. J. Williams v roce 1964. Patří mezi nejrychlejší třídicí algoritmy s minimálními nároky na pomocnou paměť, i když pro praktické účely nebývá používán tak často jako Mergesort nebo Quicksort.

Scéna: *Heapsort v haldě*

Podstata heapsortu již byla zmíněna v kapitole o haldě. Do původně prázdné haldy se nejprve operací vkládání vloží v libovolném pořadí prvky tříděné posloupnosti a pak se z ní odebírají v setříděném pořadí (od nejmenších k největším) operací vyjímání minima.

V této scéně je halda zobrazena ve své obvyklé stromové reprezentaci a je možno ji krokovat detailně i po jednotlivých voláních operací přidání a vyjmutí a animovat se zastavením po ukončení jednotlivé operace nebo až po ukončení výpočtu.

Při přidávání se prvky odebírají z pole pod haldou a po naplnění haldy jsou zase do tohoto pole z haldy odebírány. Z důvodů, které uvidíte v následujících scénách, budou do pole přidávány zprava doleva, takže budete-li výslednou posloupnost prohlížet zleva doprava, bude klesající (nebo - pokud se některá čísla v posloupnosti vyskytují opakovaně - bude nerostoucí).

Je možné si vyžádat, aby hodnoty v poli byly znázorňovány i graficky sloupečky nad příslušnými místy v poli.

Scéna: *Heapsort ve zkosené haldě*

V této scéně jsou prvky z pole do haldy odebírány zleva doprava a podobně jako v předchozí scéně jsou přidávány zprava doleva. Prvky, které jsou v poli

a nikoli v haldě jsou stále v souvislé oblasti, která se dotýká pravého okraje pole.

Halda je zobrazena tak, jak byla ukázána v kapitole Halda ve scéně Číslování vrcholů haldy při volbě **Halda a pole.** Halda je zkosená a prázdná místa v poli korespondují vzájemně jednoznačně s vrcholy haldy. Jinak ale vše probíhá stejně jako v předchozí scéně.

Scéna: *Heapsort v poli*

Scéna je opakováním předchozí, ale prvky tříděné posloupnosti, které byly přidány do haldy, jsou zobrazeny nejen číslem v některém vrcholu haldy, ale také číslem a volitelně i sloupečkem v tom místě pole, které je pod vrcholem, ve kterém se prvek nachází. Prvky tříděné posloupnosti, které se nacházejí v haldě (a současně i v poli), jsou znázorněny zelenými sloupečky, kdežto prvky, které jsou jen v poli, ale mimo haldu, jsou modré. Prohazování čísel uložených v haldě je animováno nejen v haldě samotné, ale i v poli pod ní.

Až vám činnost Heapsortu bude jasná, zatrhněte **Samotné pole** a zkuste výpočet znovu. Halda se už jako strom nebude zobrazovat, zůstanou jen prvky v poli, které se mezi sebou prohazují. Odlišení prvků o nichž víme, že jsou v haldě, je na obrazovce provedeno barvou, v programu si budeme vést proměnnou, která říká kolik je v haldě aktuálně prvků, neboli kam až halda dosahuje.

Povšimněte si, že přidání prvku do haldy (než začne probublávání) se na prvcích pole nikterak neprojeví, zůstávají ve svých polohách a pouze se zvětší oblast pole, která představuje haldu (tedy v appletu jeden sloupeček zezelená a v programu se inkrementuje počitadlo počtu vrcholů haldy).

Vyjmutí minimálního klíče z haldy, přesunutí prvku z konce haldy do kořene a vynechání vrcholu na konci haldy, což je úvodní krok vynechávání minima, se zase provede tak, že se prohodí prvky na levém konci haldy (minimum) a na pravém konci haldy a pak se hranice haldy posune o jednu pozici doleva. To znamená, že místo, kam bylo přesunuto minimum, se octne mimo haldu.

Je-li zatrženo **Samotné pole,** pak již scéna ukazuje Heapsort přesně tak, jak bývá obvykle programován, aniž by byla viditelně zobrazena halda, která představuje logické schéma algoritmu.

Kapitola 10

Bubblesort

Bubblesort neboli bublinkové třídění je jednoduchá a oblíbená metoda třídění, která ale obvykle zdaleka nedosahuje výpočetní rychlosti algoritmů, které byly uvedeny v předchozích kapitolách.

Bubblesort by měl být používán pouze ve dvou případech: jestliže je tříděná posloupnost opravdu velmi krátká a nebo je dlouhá, ale blízká setříděné posloupnosti.

Jestliže například je na zcela setříděnou posloupnost aplikován Mergesort, výpočet bude jen o málo rychlejší než když je tříděna zcela chaotická posloupnost. Mergesort postupuje systematicky a nedívá se příliš, s jakými čísly pracuje. Naopak Bubblesort, aplikován na setříděnou posloupnost, pouze zkontroluje, že není co třídit a skončí a pokud je posloupnost skoro setříděná a má výrazně monotónní charakter, pak provede pouze malé množství záměn a velmi brzo skončí. Chaotickou posloupnost ale zpracovává daleko déle než algoritmy z předchozích scén a výpočetní doba pro zpracování opačně setříděné posloupnosti je zoufalá.

Scéna: *Úvod*

Na rozdíl od předchozích kapitol je základní zobrazení tříděné posloupnosti takové, že posloupnost je nakreslena shora dolů a její prvky jsou znázorněny jako vodorovné sloupečky, nalevo se opírající o svislou základovou přímku.

Bubblesort třídí posloupnost obsahující n čísel v n fázích. V k-té fázi se předpokládá, že prvních $k-1$ prvků posloupnosti je již setříděno a do této posloupnosti přidáme k-tý prvek a to na místo, na které patří.

Stav výpočtu je znázorněn barevně. V k-té fázi je $k-1$ setříděných prvků v horní části posloupnosti zbarveno žlutě, zpracovávaný k-tý prvek červeně a zbývající prvky, se kterými algoritmus ještě nepracoval, jsou modré.

Fáze se provádí tak, že se červený aktivní prvek nechá "vybublat" do pozice, do které patří svou velikostí. Tento pohled na výpočet vedl jak k názvu

Bubblesort (v překladu bublinkové třídění), tak k tomu, že je posloupnost je znázorňována svisle (protože bublání je přece pohyb směrem vzhůru).

Jak jste již jistě poznali, bublání červeného aktivního prvku znamená, že je prohazován s prvkem bezprostředně nad ním, dokud se nedostane buď zcela nahoru a nebo bezprostředně pod prvek s menší hodnotou.

Vypnutí volby **Svisle** způsobí, že posloupnost je znázorňována zleva doprava, tedy obdobně jako v předchozích kapitolách; průběh výpočtu je ale zcela stejný, jen prvky "bublají" doleva.

Scéna: *Nejlepší případ*

Tato scéna se od předchozí odlišuje pouze tím, jak je volena vstupní posloupnost. Při zatržené možnosti **Nejlepší** je sestrojena tak, že je monotónní (způsobem který je požadován, tedy rostoucí, resp. neklesající směrem shora dolů nebo zleva doprava). Pokud je tato volba vypnuta, je posloupnost podobná, ale obsahuje několik chybně zařazených prvků.

Projděte si výpočet a zjistíte, že je v tomto případě velmi rychlý.

V této a následujícího scénách Algovize počítá počet porovnání a prohození a zobrazuje jej v horním pravém rohu okna spolu s čísly n a $n^2/2$ pro porovnání (n je počet prvků posloupnosti a $n^2/2$ je pak počet neuspořádaných dvojic prvků, které by mohly být porovnávány).

Zde vidíme, že počet porovnání je roven nebo blízký číslu n a počet prohození roven nebo blízký 0.

Scéna: *Nejhorší případ*

V této scéně je práce algoritmu ilustrována stejně jako v předchozích dvou, ale při zatržené možnosti **Nejhorší** je sestrojena tak, že je setříděna přesně opačně, než je požadováno, tedy klesající, resp. nerostoucí směrem shora dolů nebo zleva doprava). Pokud je tato volba vypnuta, je posloupnost podobná, ale obsahuje několik prvků, které tomuto pravidlu nevyhovují.

Projděte si výpočet a zjistíte, že je v tomto případě tak pomalý, jak jen může algoritmus pro třídění být - v závislosti na volbě je každý nebo skoro každý prvek porovnán a prohozen s každým jiným. Počitadlo operací ukazuje čísla blízká $n^2/2$.

Pro takto sestrojené posloupnosti je Bubblesort výrazně pomalejší než algoritmy z předcházejících tří kapitol a i než většina dalších běžně používaných metod.

Scéna: *Průměrný případ*

V této scéně je algoritmus aplikován na náhodnou posloupnost, jak tomu již bylo v první scéně. Nyní se soustřeďte na počet provedených operací.

Projděte si výpočet a zjistíte, že i v tomto případě bublají prvky velmi dlouho. Podíváme-li se na to, přes kolik úrovní prvek bublal, jsou všechny

hodnoty od 0 až po vzdálenost k hornímu prvku posloupnosti zhruba stejně časté, takže v průměru je počet přebublaných úrovní asi polovina nejhorší možné hodnoty.

Pro takto sestrojené posloupnosti je tedy Bubblesort výrazně pomalejší než algoritmy z předcházejících tří kapitol a i než většina dalších běžně používaných metod. Lze ukázat, že (jak bylo naznačeno) pro náhodnou posloupnost je počet operací obvykle blízký polovině nejhoršího možného počtu $n^2/2$. Ověřte si, zda tomu tak je.

Na rozdíl od Quicksortu, který také v základní variantě může být pomalý, ale obvykle je velmi rychlý, Bubblesort je tedy nejenom v nejhorším případě, ale i obvykle velmi pomalý.

Kapitola 11

Hledání mediánu

Jsou dána čísla $a_1, \ldots, a_n$ a číslo k. Cílem je nalézt k-tý prvek, což je číslo, které by bylo na k-té pozici, pokud bychom zadaná čísla setřídili. Nejčastější případ je hledání mediánu, což je k-tý prvek pro $k = \lfloor n/2 \rfloor$. Poznamenejme, že čísla n a k lze nastavit na ovladači a nový soubor čísel vytvořit knoflíkem **Nový vstup.**

Úloha hledání k-tého prvku nebo mediánu se snadno vyřeší podle definice tím, že se čísla setřídí podle velikosti a odpočítá se k-té. Cílem tohoto appletu je ukázat, že tento postup je ale plýtváním časem a úlohu lze vyřešit obvykle podstatně rychleji.

Dá se dokázat, že pokud jediná povolená nebo možná operace s čísly je jejich porovnání a popřípadě záměna porovnávaných čísel, pak pro setřídění n čísel je v nejhorším i průměrném čase potřeba alespoň $\log_2(n!)$ porovnání, což je více než $n \log_2 n - n \log_2 e$.

Za stejného omezení lze ale medián nebo obecněji k-tý prvek pro libovolné k nalézt i v nejhorším případě v lineárním čase, použijeme-li způsob uvedený v této kapitole nebo některé jeho zdokonalení.

Scéna: *Základní algoritmus*

Tato scéna popisuje hledání k-tého prvku nebo mediánu jednoduchým algoritmem, který je velmi podobný Quicksortu. Je to vlastně opravdu algoritmus Quicksort, ve kterém vynecháváme práce, které není nutno provést. Prvky posloupnosti jsou přitom nakresleny rozhozené po ploše okna. Algoritmus spočívá v opakovaném provádění následující posloupnosti kroků:

Krok: *Určení pivota*

První krok algoritmu je určení pivota. To spočívá ve výběru libovolného čísla z tříděného souboru. Pivot je označen červeně. ◇

Krok: *Rozdělení podle vztahu k pivotu*

Následující krok výpočtu je rozdělení čísel do tří skupin podle jejich vztahu k pivotu. Čísla menší než pivot se označí žlutě, čísla větší než pivot zeleně a nakonec čísla stejná jako pivot růžově (výjimkou je box, ze kterého byl vybrán pivot; ten je stále červený). ⋄

Krok: *Určení velikosti skupin*

V tomto kroku se skupiny čísel pro přehlednost seřadí a znázorní odděleně a u každé se určí, kolik čísel v ní leží. Tato činnost se obvykle provádí současně s rozdělováním čísel do skupin, ale v appletu je v zájmu lepší srozumitelnosti oddělena. ⋄

Krok: *Odstranění nepotřebných skupin*

Když je známo, jak jsou skupiny čísel velké, je velmi jednoduchá spočítat, ve které skupině se k-tý prvek nachází a kolikátý v ní bude.

Jestliže hledáme k-tý nejmenší prvek a skupiny prvků menších, stejně velkých, resp. větších než pivot mají n_1, n_2, resp. n_3 prvků, pak
pokud je $k \leq n_1$, budeme v dalším hledat k-tý nejmenší prvek mezi n_1 prvky menšími než je pivot,
pokud je $n_1 < k \leq n_1 + n_2$, je k-tý nejmenší prvek roven pivotu a výpočet končí, a
pokud je $n_1 + n_2 < k$, budeme v další hledat $(k - n_1 - n_2)$-tý nejmenší prvek mezi n_3 prvky většími než je pivot.

Pokud nenastává druhý případ, pak skupina se kterou se pokračuje ve výpočtu je určitě menší než původní (minimálně neobsahuje pivota) a proto postup po jisté době musí skončit a dát správný výsledek.

Volba pivota ale je důležitá pro rychlost výpočtu. V nejhorším případě má prostřední skupina 1 prvek (pivot) a první nebo třetí skupina prázdná. Pak skupina, kterou se pokračuje, bude mít jen o 1 prvek méně členů a počet iterací postupu bude roven n, což povede k celkově kvadratickému počtu provedených porovnání.

Pokud by naopak první a třetí skupiny byly stejně velké (nebo se jejich velikost lišily o 1, je-li $n - 1$ liché číslo), pak bude $n_1, n_3 \leq n/2$ a při každé další iteraci bude velikost zpracovávané skupiny poloviční a skončíme po logaritmickém počtu iterací.

Abychom ale zaručeně zvolili pivota tak, aby se nepivotové prvky rozdělily do přesně stejných skupin, potřebovali bychom za pivota volit medián a to je to, co teprve algoritmem hledáme.

V následující scéně se ale ukáže, že stačí umět určit přibližný medián (zatím neříkám, co tím přesně myslím) a to je daleko jednodušší. ⋄

Scéna: *Lineární algoritmus*

Tato scéna ukazuje algoritmus, který medián nalezne v lineárním čase. Jedná se o variantu předchozího algoritmu, která ale pivota v každém kroku vybírá velmi speciálním způsobem.

Krok výběru pivota z předchozí scény je zde nahrazen následující posloupností akcí

Krok: *Rozdělení do pětic*

První krok výběru pivota je rozdělení souboru čísel do pětic. Zdůvodnění volby čísla pět se dočkáte až při analýze algoritmu. Případnou neúplnou pětici nebudeme brát v úvahu. ◇

Krok: *Nalezení mediánu pětic*

V každé pětici určíme medián. Na obrazovce prvky každé pětice přemístíme tak, aby medián byl uprostřed a menší (resp. větší) dva prvky byly nad ním (resp. pod ním). Pořadí prvků horní i dolní dvojice není důležité; pětice tedy nemusíme úplně uspořádat. ◇

Krok: *Určení mediánu mediánů*

Určíme medián souboru mediánů pětic a prohlásíme jej za pivota. Na obrazovce pětice přesuneme tak, aby se medián mediánů ocitl uprostřed a aby pětice jejichž medián je menší (resp. větší) než medián mediánů byly nalevo (resp. napravo) od pětice obsahující pivota. Pořadí pětic ve skupině nalevo i ve skupině napravo není důležité. Ve skutečném programu takové přesuny nejsou nutné, stačí se zabývat souborem mediánů, ale v appletu jsou užitečné pro vizualizaci myšlenky algoritmu. ◇

Vyhněte se prosím časté chybě povrchních studentů. Medián mediánů pětic skoro nikdy *není* mediánem celého souboru. Jak bude podrobněji uvedeno v následujících odstavcích, dá se pouze ukázat, že asi 30 procent prvků je menších nebo rovných tomuto pivotu (a tedy nepatří do třetí skupiny podle předchozí scény) a asi 30 procent prvků je větších nebo rovných pivotu (a tedy nepatří do první skupiny podle předchozí scény). Medián mediánů lze tedy považovat za velmi přibližnou náhražku mediánu celé posloupnosti, která ale jako pivot je pro rychlost algoritmu dostatečná.

V okamžiku určení mediánu mediánů se aktivují dva navzájem se vylučující checkboxy, které slouží pro výklad velikosti skupin vzniklých rozdělením podle pivota určeného jako medián mediánů pětic. Kliknutím na levý z nich se na pozadí zobrazí fialový obdélník.

Prvky, které fialový obdélník určuje, jsou určitě menší nebo rovné pivotu. Spadají totiž do pětic, jejichž mediány jsou menší nebo rovné pivotu (medián mediánů pětic) a v každé pětici představují medián a dva prvky nad ním (tedy menší než medián uvedené pětice). Tento vztah ilustrují také šipky, které se

objeví současně s fialovým obdélníkem a které směřují vždy k prvku menšímu nebo rovnému.

Nejvýše čtyři prvky se nepodařilo zařadit do pětic, takže pětic je alespoň $(n-4)/5$. Pětic, jejichž medián je menší nebo roven pivotu (mediánu mediánů), je alespoň polovina tohoto počtu, tedy nejméně $(n-4)/10$. Z každé takové pětice obsahuje fialový obdélník 3 prvky, tedy pokrývá alespoň $3(n-4)/10 = 3n/10 - 12/10$ prvků. Proto skupina čísel větších než pivot nemůže mít více než $7n/10 + 12/10$ prvků.

Podobně druhý checkbox aktivuje bílý obdélník, obsahující zaručeně jen čísla větší nebo rovné pivotu a proto čísla menší než pivot musí ležet vně obdélníka a může jich také být nejvýše $7n/10 + 12/10$ prvků.

Ukážeme nyní, že existuje konstanta c taková, že právě popsaný algoritmus určí k-tý prvek po nejvýše cn krocích, kde n je délka zpracovávané posloupnosti. Označme jako $T(n)$ počet kroků výpočtu, dostačujících i v nejhorším případě k nalezení k-tého prvku z n čísel.

Počet kroků nutných pro určení mediánu pěti čísel je omezen nějakou konstantou c_1. Pokud je určen pivot, pak rozdělení čísel do 3 skupin podle jejich vztahu k pivotu a určení velikosti těchto skupin vyžaduje jistě nejvýše $c_2 n$ kroků pro nějakou konstantu c_2.

Nyní dokážeme indukcí, že $T(n) \leq cn$, kde

$$c = max(T(1), T(2)/2, \ldots, T(23)/23, 4c_1 + 20c_2).$$

Jestliže je $n < 24$, pak $T(n) = (T(n)/n) \cdot n \leq cn$.

Pokud $n \geq 24$, pak $T(n)$ zahrnuje:
určení mediánu nejvýše $(n-4)/5$ pětic, které si vyžádá nejvýše $c_1(n-4)/5 \leq c_1 n/5$ kroků;
určení mediánu mediánů pětic, kterých je nejvýše $(n-4)/5$; jelikož toto číslo je určitě menší než n, pak podle indukčního předpokladu k určení mediánu mediánů není třeba více než $c(n-4)/5 < cn/5$ kroků;
rozdělení prvků posloupnosti do 3 skupin podle jejich vztahu k pivotu rovnému mediánu mediánů a určení skupiny, ve které se bude dále pokračovat, což vyžaduje podle výše uvedeného nejvýše $c_2 n$ kroků;
pokud nebylo zjištěno, že medián posloupnosti je roven pivotu, pokračuje se určením prvku stanoveného pořadí ve skupině nejvýše $7n/10 + 12/10$ prvků; jelikož toto číslo je pro $n \geq 8$ menší nebo rovno $n-1$, pak se tato operace podle indukčního předpokladu zvládne v nejvýše $c(7n/10 + 12/10)$ krocích.

Platí tedy

$$T(n) \leq \frac{c_1 n}{5} + \frac{cn}{5} + c_2 n + \frac{7cn}{10} + \frac{12c}{10} = \frac{9cn}{10} + \frac{2c_1 n + 10c_2 n}{10} + \frac{12c}{10} \leq cn,$$

neboť $2c_1 n + 10c_2 n = (4c_1 + 20c_2)n/2 \leq cn/2$ a
jelikož $n \geq 24$, pak také $12c \leq cn/2$.

Kdybychom dělili prvky do trojic pak fialový a bílý obdélník by obsahovaly zhruba $n/3$ prvků, takže velikost skupin by byly nejvýše $2n/3$, ale museli bychom určovat medián $n/3$ mediánů trojic, takže bychom dostali pro nějaké konstanty c_1 a c_2 zhruba

$$T(n) \leq \frac{c_1 n}{3} + \frac{cn}{3} + c_2 n + \frac{2cn}{3} = \frac{3cn}{3} + \frac{c_1 n + 3c_2 n}{3} = cn + \frac{c_1 n + 3c_2 n}{3},$$

ale z toho by nerovnost $T(n) \leq cn$ nevyplynula. U čtveřic není medián uprostřed čtveřice a proto by algoritmus také nefungoval. Z tohoto důvodu jsme volili rozdělení do pětic. Rozdělení do sedmic nebo vyšších lichých skupin je také možné a může být i výhodnější, ale u příliš velkých skupin je už zase obtížnější hledat jejich medián. Otázkou, jak velké skupiny jsou nejvhodnější, ze zabývat nebudeme, pokud bychom chtěli dosáhnout co nejnižší konstanty v lineárním odhadu, je toho možné dosáhnout poněkud jiným typem algoritmů, které ale pro jejich přílišnou složitost zde neuvádím.

Kapitola 12

Bitonické třídění

Scéna: *Úvod*

Tato scéna je jen ilustrací a ukázkou výsledného třídicího obvodu, přejděte prosím hned dále.

Scéna: *Komparátor*

Komparátor je obvod se dvěma vstupy a dvěma výstupy, které přenášejí čísla. Komparátor porovná čísla, která do něho vstoupí shora; pokud levé číslo je menší nebo rovné pravému, pak je takto odešle dál z výstupů na své spodní straně, tedy levý vstup převede na levý výstup a pravý vstup na pravý výstup. Pokud je levé vstupní číslo je větší než pravé, pak je zamění, tedy levý vstup zavede na pravý výstup a pravý vstup na levý výstup.

V řádcích na ovladači, označených **N1** a **N2** zadejte 2 vstupy pro komparátor (malá celá nezáporná čísla) a sledujte odpovídající výstupy. Je také znázorněno, zda komparátor převádí čísla v jejich původním pořadí nebo je prohazuje.

Scéna: *Třídící obvod s komparátory*

V této scéně je nakreslen obvod pro třídění 8 čísel. Model je následující: Libovolných 8 čísel se přivede na vstupy v horní části obrazovky a nechá postupovat směrem dolů po černých čarách představujících vodiče. Obdélníky představují *komparátory*, viz minulá scéna. Obvod je navržen tak, aby na spodní straně obrazovky se čísla objevila setříděna zleva doprava jako neklesající posloupnost.

Vstupní čísla je možno zvolit knoflíkem **Nový vstup** a postup třídění sledovat graficky. Červené sloupečky pod jednotlivými vodiči graficky znázorňují hodnoty čísel v příslušných vodičích v úrovni označené červenou vodorovnou

čarou s trojúhelníčky po stranách, kterou je možno posouvat nahoru a dolů knoflíky bloku označeného **Výpočet.** Červenou čáru je možno si představovat jako hranici, po kterou již postoupilo třídění v obvodu. Nad červenou čárou je také u každého komparátoru znázorněno, zda je sepnut přímo nebo křížem.

Předpokládáme-li, že komparátory mohou pracovat současně, a že zpracování vstupů v komparátoru trvá jistý konstantní časový interval, je doba potřebná k setřídění čísel rovna počtu vodorovných vrstev, do kterých je možno obvod rozdělit tak, aby vodiče byly vedeny směrem dolů.

Obvod typu, znázorněného na obrazovce je vlastně varianta bublinového třídění a není problém jej nakreslit pro libovolný počet vstupů. Zkuste si změnit číslo na ovladači, zobrazí se obvod příslušné velikosti. Je ovšem vidět, že hloubka obvodu a tedy i doba, za kterou by se při praktické realizaci ustálily správné hodnoty na výstupu, roste lineárně s počtem vstupů. V této kapitole budeme řešit otázku, zda je z komparátorů možné sestavit obvod jinak, aby jeho hloubka byla menší.

Ajtai, Komlós a Szemerédi dokázali, že obvod je možno sestrojit tak, že pro jistou konstantu C nebude mít více hladin než $C \log n$, kde n je počet vstupů. Konstanta C jejich obvodu je ale astronomicky vysoká a není známo, zda je možné ji snížit a navíc astronomicky vysoká je také logická složitost konstrukce. Ačkoliv byl článek publikován v roce 1983, nebyl dosud zásadně zlepšen. V následujících scénách ale ukážeme relativně jednoduchou konstrukci, pocházející od K. E. Batchera z roku 1968, která pro $n = 2^k$ umožňuje sestrojit třídicí obvod s $k(k+1)/2$ vrstvami, což je přibližně $0.5 \log_2^2 n$.

Scéna: *Bitonická posloupnost*

Než se pustíme do konstrukce Bakerova obvodu, avizovaného v předchozí scéně, musíme si probrat pojem *bitonická posloupnost.* Bitonická posloupnost je například posloupnost která napřed roste a potom klesá. Jiný příklad je posloupnost, která napřed klesá a potom roste. Odtud také název - bi-tonus znamená něco jako dva směry.

Obecná definice bitonické posloupnosti je ale složitější. Pokud si ji znázorníme graficky a označíme si její minimum a maximum, pak od minima můžeme postupovat směrem k maximu přímo a nebo také tak, že od minima jdeme opačným směrem až na kraj oblasti, na které je posloupnost nakreslena, přeskočíme na druhý okraj a pak již jdeme přímo k maximu. Stisknutím knoflíku **1. cesta** nebo **2. cesta** spustíte pohyb kurzoru, který vám cesty ukáže. Posloupnost je bitonická právě tehdy, když při postupu obojím způsobem hodnoty prvků posloupnosti rostou nebo alespoň neklesají. Znamená to, že při animaci postupu 1. nebo 2. cestou výška zvýrazněného sloupečku nesmí klesnout.

Bitonickou posloupnost si často znázorňujeme na kružnici. Lineární zobrazení posloupnosti představuje hřeben. Jestliže se hřeben ohne do kruhu tak,

aby paprsky trčely ven a konce se spojily, dostaneme výhodný způsob grafického znázornění, kde se smazává rozdíl mezi první a druhou cestou z minima do maxima, protože odpadá přeskok z jednoho konce intervalu na druhý. Stiskněte knoflík **Transformace** pro schématickou animaci transformace z lineární na cyklickou representaci.

Zkoušejte si nějaký čas generovat nové posloupnosti, systém předkládá občas bitonickou posloupnost (kreslena červeně) a občas pro kontrast posloupnost zcela náhodnou (kreslena modře).

Knoflíkem **Otočení** se posloupnost cyklicky posune o tolik pozic, kolik je udáno v poli vedle knoflíku. Poznamenávám, že cyklické otočení (doprava) posloupnosti $a_1, \ldots, a_n$ o jednu polohu je posloupnost $a_n, a_1, \ldots, a_{n-1}$ (vše se posune o 1 polohu doprava, jen nejpravější prvek skočí na uprázdněný levý konec). Posun o k poloh se dostane jako k-násobné opakování posunu o 1 polohu. Posun o k poloh doleva je totéž jako posun o $n - k$ poloh doprava, posun o n poloh nic nemění.

Základní pozorování je, že cyklický posun bitonické posloupnosti dává zase bitonickou posloupnost; sledujte obrázek a zkuste si to dokázat.

Další způsob jak charakterizovat bitonické posloupnosti je, že že jsou to právě ty posloupnosti, které vzniknou cyklickým posunem z posloupnosti, která napřed roste a pak klesá.

Scéna: *Bitonické třídění*

Třídící obvod podle K. E. Bakera se obvykle nazývá *bitonický obvod*. Vysvětlit konstrukci bitonické třídičky je jednoduché. Provedeme to rekurzivním způsobem. Poznamenávám, že předpokládáme, že počet vstupů je mocnina dvojky. Pokud by počet vstupů byl obecné číslo, rozšíříme jej na nejbližší vyšší mocninu 2 (méně než dvojnásobek) a na přidané vstupy přivedeme hodnoty $+\infty$, aby se nemíchaly s ostatními čísly.

Poznamenejme jen, že budeme používat komparátory dvou typů: vzestupné, které dávají větší číslo na pravý výstup a sestupné, které dávají větší číslo na levý výstup. Odlišeny budou barevně, větší číslo patří na stranu nakreslenou tmavě, menší číslo na stranu nakreslenou světle. Stejně tak budeme potřebovat umět sestrojit třídicí obvod vzestupný i obvod sestupný. Podána bude konstrukce obvodu vzestupného, je ale zřejmé, že obvod sestupný se z něho získá záměnou typu u všech jeho komparátorů.

Pro malé hodnoty n je konstrukce triviální: pro $n = 1$ není co třídit, pro $n = 2$ se použije jediný komparátor.

Předpokládejme, že je n vyšší, výchozí obrázek na obrazovce má $n = 32$. Třídící obvod je nejprve zobrazen jako černá skříňka. Můžete si ověřit, že správně funguje (knoflíky bloku **Výpočet),** ale není vidět proč správně funguje ani jak je konstruován. Konstrukce obvodu, tedy rozbalování černé

skříňky, se krokuje knoflíky bloku **Konstrukce** a spočívá v opakovaném provádění následujících kroků:

Krok: *Redukce na třídění bitonické posloupnosti*

Sestrojíme si rekurzivně obvod třídící vzestupně s $n/2$ vstupy, který použijeme na setřídění levé poloviny vstupů - je to černý obdélník vlevo nahoře. Potom rekurzivně sestrojíme obvod třídící sestupně, opět s $n/2$ vstupy, a použijeme jej pro setřídění pravé poloviny vstupů - je to černý obdélník vpravo nahoře.

Výstupy těchto dvou menších obvodů dají dohromady bitonickou posloupnost, která napřed roste a potom klesá. Pak sestrojíme obvod s n vstupy, který umí třídit jenom bitonické posloupnosti a tím získanou posloupnost setřídíme. Tento obvod je znázorněn jako široký šedivý obdélník dole. V předchozí scéně jsme viděli, že bitonická posloupnost je výrazně jednodušší než obecná posloupnost a proto není divu, že její setřídění se ukáže jako poměrně jednoduchý úkol. Vnitřní konstrukce tohoto bloku bude ukázána v dalším kroku.

Zkuste si třídění krokovat; applet ukáže kromě výchozí a konečné posloupnosti ještě mezivýsledek, který je zjevně bitonická posloupnost. ◇

Krok: *Separační vrstva*

Konstrukce obvodu pro setřídění bitonické posloupnosti je ukázána v dalším kroku konstrukce. Všech n vstupů napřed přivedeme do zeleně označeného bloku, nazývaného *separátor*, jehož vnitřní konstrukci nebudeme zatím probírat, a jehož funkce je následující:

jsou-li výstupy zeleného bloku zleva doprava čísla $x_1, \ldots, x_n$, pak pro libovolné i, j takové, že $i < n/2 \leq j$ platí, že $x_i \leq x_j$ a navíc posloupnosti $x_1, \ldots, x_{n/2}$ i $x_{n/2+1}, \ldots, x_n$ jsou bitonické. Jinými slovy nalevo přijdou menší čísla, napravo větší a navíc levá i pravá polovina jsou zase bitonické.

Nyní je zřejmé, že stačí odděleně setřídit levou polovinu výstupů separátoru třídícím obvodem s $n/2$ vstupy, který umí třídit pouze bitonické posloupnosti, stejně tak setřídit i pravou polovinu výstupů a tím se dostane setříděná posloupnost. Tyto činnosti se provedou dvěma šedivými bloky v levé a pravé spodní části schématu. Tyto bloky jsou navrženy pro třídění stejným směrem jako má třídit konstruovaný blok.

Až na konstrukci separátorů tedy již umíme celý třídicí obvod zkonstruovat rekurzivní aplikací výše uvedených postupů.

Zkuste si znovu krokovat postup třídění a sledovat formu posloupnosti před a po separačním bloku. ◇

Krok: *Konstrukce separátoru*

Separátor obsahuje jednu vrstvu zahrnující $n/2$ komparátorů. Pro $i = 1, \ldots, n/2$ porovnává i-tý separátor vstupy s pořadovými čísly i a $i + n/2$ a své výstupy převede na výstupy bloku se stejnými pořadovými čísly.

Konstrukce separátoru je tedy velmi jednoduchá, není ale jednoduché dokázat, že takto sestrojený blok má výše požadovanou funkci. Tomu budou věnovány následující scény. ◇

Dokončete si celou konstrukci třídicího obvodu; v každé etapě konstrukce je obvod plně funkční a je proto možné zadávat nové vstupy a krokovat průběh třídění. Nejprve se postupně dokončí rozvinutí spodních šedých bloků, určených pro třídění bitonických podposloupností. Potom se obdobným způsobem rozvinou třídící obvody pro poloviny vstupů v horní části schématu.

Nyní je také vidět, proč definice bitonické posloupnosti je tak komplikovaná. Před prvním separátorem je posloupnost, která je "velmi bitonická". Nejprve stoupá přesně do poloviny, potom v druhé polovině klesá. Na výstupu hlavní separační vrstvy jsou bitonické posloupnosti dvě: první zase stoupá a pak klesá, druhá ale již má obrácený průběh - napřed klesá a pak stoupá. Navíc ale úseky stoupání a úseky klesání nejsou obecně stejně dlouhé. To má za následek to, že na výstupu další separační vrstvy už se objeví čtyři čtvrtinové bitonické posloupnosti, které ale už vypadají zcela obecně; může se například stát, že napřed stoupá, pak klesá a pak stoupá, ale ne výše než začala (i tak může bitonická posloupnost vypadat). "Mira bitoničnosti" posloupností stále klesá a definice postihuje hranici, pod kterou již nikdy neklesneme ani u největších obvodů.

Scéna: *Půlení bitonické posloupnosti*

Než se budeme moci pustit do důkazu vlastnosti separačního bloku, musíme si dokázat jednu důležitou vlastnost bitonické posloupnosti, kterou naformulujeme pro cyklické znázornění:

kružnici nesoucí členy bitonické posloupnosti je možno rozetnout přímkou procházející středem kružnice na dvě části tak, že maximum v jedné části je menší nebo nejvýše rovno minimu druhé části.

Klepnutím na knoflík **Rozpůlení** se takové rozdělení ukáže a to i na cyklickém i na lineárním znázornění posloupnosti. Malé prvky jsou nyní zobrazeny žlutě, velké zůstávají červené.

Je možné opakovaně konstruovat nové bitonické posloupnosti a v nich si nechávat zobrazit rozpůlení. Důkaz existence rozdělení pro libovolnou bitonickou posloupnost ukážeme v následující scéně.

Zkuste si ještě vlastnost naformulovat v řeči lineárního zobrazení posloupnosti.

Scéna: *Důkaz existence půlení*

Popíšeme nyní proces, který pro každou bitonickou posloupnost sestrojí požadované půlení. Proces bude popsán dvakrát; jednou jak se jeví v lineární representaci, jednou jak vypadá na kružnici.

Představme si, že se na obrázek s lineární representací bitonické posloupnosti napouští voda. Sloupeček, který se celý ponoří pod hladinu znázorněnou tenkou bílou čarou, se utopí a zežloutne. Z vlastností bitonické posloupnosti plyne, že utopenci tvoří stále souvislou skupinu, mezi kterou není žádný dosud žijící sloupec. Pokud ovšem utopená skupina dosáhne k jednomu kraji, začne přibývat i od druhého kraje.

V cyklické representaci vytváříme výseč, která odděluje malé žluté prvky od velkých červených. Výseč obsahuje na začátku jenom minimum posloupnosti a postupně se k němu přidává vždy ten červený prvek sousedící s hranicí výseče, který je menší. Jelikož je posloupnost bitonická, tedy od minima k maximu je oběma směry (ve směru i proti směru hodinových ručiček) neklesající, je maximum žlutých prvků ve výseči nejvýše rovno minimu červených prvků mimo ni.

Postup pokračuje dokud se neutopí (neboli nedostane se do výseče) přesně polovina prvků. V ten okamžik běží ramena výseče v opačných směrech a tedy dohromady představují přímku. Nápis v horní části okna napovídá, jak jsme s důkazem pokročili.

Scéna: *Komparátory separační vrstvy*

Bitonická posloupnost je nyní znázorněna i se sestrojeným rozpůlením. Navíc je přikresleno zařízení, představující jeden komparátor separační vrstvy, který porovnává jeden vstup z levé poloviny vstupů s odpovídajícím vstupem z pravé poloviny. "Odpovídající" přitom znamená "vzdálený o $n/2$". Knoflíkem **Vpravo** nebo **Vlevo** zařízením můžeme posouvat do poloh dalších komparátorů separační vrstvy bitonického třídicího obvodu.

Komparátor je znázorněn i v cyklické representaci; tam je jeho poloha obzvláště přehledná, neboť porovnává prvky vzdálené o $n/2$ a tedy na kružnici protilehlé.

V cyklické representaci je nyní jasně vidět, proč platí základní vlastnost separační vrstvy: každý komparátor bez ohledu na svou polohu vždy porovnává jeden žlutý prvek s jedním červeným. Na svůj levý výstup, tedy do levé poloviny výstupů separační vrstvy, proto vždy dá žlutý prvek a na svůj pravý výstup, tedy do pravé poloviny výstupů, dá vždy červený prvek. V levé polovině výstupů se proto objeví výhradně žluté, menší, prvky a v pravé polovině zase jenom červené, větší, prvky.

Scéna: *Funkce separační vrstvy*

Tato scéna opakuje znovu scénu předchozí, ale checkboxy v dolní části ovladače je možno způsobit, že se v lineárním zobrazení vykreslují hodnoty levého a/nebo pravého výstupu komparátoru. Knoflíkem **Zpět** je možno výstupy vymazat a nechat kreslit znovu.

Nejprve si zkuste nakreslit celou výstupní posloupnost. Pak výstupy vymažte; ukážeme si ještě proč levá a pravá poloviční výstupní posloupnost jsou zase bitonické. Nechte zatím oba checkboxy pro výstupy neaktivní.

Snadno se ověří, že každý výsek bitonické posloupnosti je zase bitonická. Proto rozetneme-li kružnici jak je znázorněno, dostaneme dvě posloupnosti, žlutou a červenou, které (v pořadí v jakém šly kolem kružnice) jsou zase bitonické. Nastavte nyní komparační zařízení tak, aby ukazovalo na jeden konec žluté posloupnosti na kružnici a aktivujte checkbox pro levý vstup.

Posouváním obrazu komparátoru přes všechny jeho polohy "navineme" žlutou bitonickou posloupnost do levé poloviny výstupů separační vrstvy. Posloupnost, která se tedy nalevo objeví, je cyklickým posunutím žlutého segmentu kružnice, který je bitonický a proto je bitonická také. Stejný důkaz lze provést i pro druhou, červenou posloupnost.

Scéna: *Bitonický obvod jinak nakreslený*

Tato scéna ukazuje jiné nakreslení bitonického obvodu. Komparátory vždy porovnávají sousední vodiče, takže je možno je kreslit jako jednoduché obdélníky. Platíme za to tím, že "vodiče" nejsou svislé úsečky, ale dosti klikatě se proplétající lomené čáry. Ověřte si, že tento obvod a obvod z předchozích scén (který se vykreslí po deaktivaci checkboxu **Alternativní)** jsou identické, i když na první pohled vypadají dosti odlišně.

Část III

Grafové algoritmy

Tato část knihy popisuje algoritmy pro řešení tří základních úloh diskrétní optimalizace: hledání nejkratší cesty, minimální kostry a největšího toku v dané síti. Tyto úlohy jsou považovány za zlatý fond teorie algoritmů a alespoň některé výpočetní postupy naleznete v každé učebnici této problematice věnované.

Začneme pomocnou kapitolou o prohledávání grafů, speciálně o prohledávání do šířky a do hloubky, které představují velmi důležitou techniku, která je při návrhu grafových algoritmů velmi často používána. Jako jednoduchou aplikaci také uvedu hledání komponent a 2-souvislých komponent neorientovaného grafu. Prohledávání bude použito i dále, např. v kapitole o tocích v sítích.

Pro hledání nejkratší cesty v grafu uvedeme v další kapitole tři algoritmy různé složitosti a okruhu aplikovatelnosti. Bude uveden algoritmus kritické cesty pro acyklický graf, Dijkstrův algoritmus pro nezáporně hranově ohodnocený grafy a Bellman-Fordův algoritmus pro grafy bez záporného cyklu.

Dalším grafovým problémem, kterým se v této části budeme zabývat, je hledání minimální kostry grafu s ohodnocenými hranami. Bude podáno jednoduché algoritmické schéma, které úlohu řeší a několik jeho implementací.

Jedním z nejzajímavějších grafových problémů je hledání maximálního toku v síti. Pro jeho řešení bylo od roku 1962, kdy vyšla základní kniha Forda a Fulkersona o tocích v sítích, navrženo opravdu velmi velké množství algoritmů nejrůznější výpočetní rychlosti i logické obtížnosti. Budou popsány tři algoritmy.

Původní Ford-Fulkersonův algoritmus je klíčem pro pochopení většiny ostatních metod, ale sám už prakticky není využíván, protože je sice velmi jednoduchý, ale může za jistých okolností pracovat velmi dlouho (a teoreticky výpočet nemusí skončit nikdy).

Dinitzův algoritmus z roku 1970 je pěknou ukázkou jedné třídy algoritmů pro toky v sítích, která zobecňuje Ford-Fulkersonův algoritmus a hledá maximální tok jeho zlepšováním podél cest. Velmi podobný algoritmus navrhli nezávisle v roce 1972 Edmonds a Karp.

Goldbergův algoritmus z roku 1987 je založen na zcela jiném, celkem přirozeném, principu: do sítě vpustíme tak veliký tok, jaký je ze zdroje možno vyslat; část vstupní vlny se nakonec dostane do spotřebiče a ta část, která už sítí nemůže protéci, se odrazí a nakonec se vrátí zpět do spotřebiče.

Poslední kapitolka této části trochu vybočuje z výše uvedeného rámce a nachází se na pomezí algebry a teorie grafů. Ukazuje, jak je možno hledat minimální řez v grafu pomocí spektrální teorie grafů, konkrétně pomocí výpočtu vlastních vektorů incidenční matice grafu. Jedná se sice o heuristický algoritmus, který nezaručuje optimalitu řešení; tu ale nezaručuje žádný ze známých dostatečně rychlých algoritmů a je mnoho náznaků, že takový rychlý a současně optimální algoritmus ani neexistuje. Spektrální heuristiky jsou po-

važovány za jedny z nejlepších postupů, jak hledat minimální řez, ale jsou většinou informatiků považovány za pochopitelné pouze specialistům v algebraické teorii grafů. Cílem jednoduchého appletu, uvedeného v této kapitole je ukázat, že jsou naopak velmi přirozené a při vhodném pohledu na problém i snadno pochopitelné.

Kapitola 13

Prohledávání grafu

Scéna: *Obecné prohledávání neorientovaného grafu*

Prohledávání grafu je systematické prozkoumání grafu podobným způsobem, jakým by se mohla prohledávat neznámá krajina.

Při prohledávání budeme dělit vrcholy na tři skupiny: nedosažené, dosažené a probrané. Nedosažené vrcholy, označované žlutě, jsou vrcholy, ve kterých jsme ještě nebyli. Dosažené vrcholy, zbarvené zeleně, jsou vrcholy, do kterých jsme už vstoupili, ale ještě jsme neprobrali všechny možnosti, jak z nich pokračovat dále. Konečně probrané vrcholy, označené černě, jsou ty vrcholy, do kterých už jsme vstoupili a dokonce jsme i probrali všechny možnosti dalšího postupu. Barevnou výjimkou bude ten z dosažených vrcholů, který bude zvolen ke zpracování (viz dále) - jeho barva se ze zelené změní na červenou, a dále počátek prohledávání, stále tmavě modrý.

Podobně budeme dělit hrany na probrané, které budou kresleny černou barvou a neprobrané, které budou modré. Probrané hrany dále rozdělíme na užitečné a neužitečné. Hranu označíme za probranou, jestliže jsme vstoupili do jednoho jejího konce a pak jsme hranou prošli ve snaze se dostat do ještě neprobádaných míst. Ostatní hrany jsou neprobrané. Hrana se označí za užitečnou, pokud se jí prošlo do neprobraného vrcholu, tedy se objevila dosud neznámá končina, jinak je hrana neužitečná. Grafické rozlišení užitečných a neužitečných hran bude popsáno dále.

Obecný postup prohledávání je následující:

Na začátku jsou všechny vrcholy nedosažené a všechny hrany neprobrané. Potom dokud existuje alespoň jeden vrchol, který není probraný, provádíme následující činnost:

1. Pokud existuje alespoň jeden dosažený (ale neprobraný) vrchol, pak si vybereme jeden z nich, který označíme jako v a

(a) buď z v nevychází žádná neprobraná hrana a potom označíme v jako probraný

(b) nebo si vybereme jednu neprobranou hranu vycházející z v, a

i. pokud byl její druhý konec nedosažený, pak jej překlasifikujeme za dosažený a hranu označíme za probranou a užitečnou

ii. pokud byl její druhý konec dosažený nebo probraný, pak hranu označíme za probranou a neužitečnou.

2. Pokud neexistuje žádný dosažený vrchol, ale existují nedosažené vrcholy, pak si jeden z nich vybereme a označíme jej za dosažený.

Jak již bylo řečeno, pokud jsou všechny vrcholy probrané, pak výpočet končí. Je zřejmé, že v ten okamžik musí také všechny hrany být probrané, protože když je některá hrana neprobraná, pak její konce ještě nemohly být označeny za probrané vrcholy.

Probrané hrany (tedy v okamžiku ukončení prohledávání všechny hrany) jsou přirozeným způsobem orientovány tím, v jakém směru jsme je prošli. Orientace neužitečných hran pro nás nebude příliš zajímavá a proto ji nebudeme vyznačovat. Orientaci užitečných hran graficky vyjádříme tím, že jakmile je hrana klasifikována jako užitečná, objeví se pod ní tlustá bílá šipka orientovaná *proti směru*, ve kterém jsme hranou procházeli. Bílá šipka tedy pro každý vrchol ukazuje, odkud jsme do něho poprvé vstoupili. Bílé šipky také graficky odlišují užitečné hrany (černé a mají pod sebou bílou šipku) od neužitečných hran (černé bez podkladové šipky).

Vrcholy budeme také číslovat čísly $1, 2, \ldots$ v pořadí, ve kterém byly dosahovány. Toto pořadí poskytuje často užitečnou informaci o grafu. Nedosažené vrcholy očíslovány (zatím) nejsou.

Podrobněji lze výpočet popsat takto: Na začátku jsou všechny vrcholy nedosažené a proto v bodě 2 zvolíme jeden z vrcholů jako počátek prohledávání a překlasifikujeme jej na dosažený. Pak se dosažená oblast v grafu rozšiřuje tak, že pro další postup musíme vycházet z dosažené oblasti a postupovat jen hranami, kterými jsme ještě nešli.

Jestliže se po přechodu po hraně dostaneme do míst, kde jsme již byli, považujeme tento přechod za zbytečný a hranu označíme za neužitečnou, ale pokud pronikneme do panenské oblasti, prohlásíme hranu za užitečnou a nově dobyté území si označíme. Jestliže z některého vrcholu byly podniknuty výpravy všemi možnými směry, vrchol již nepřinese nic nového a je označen za probraný.

Jestliže se z počátečního vrcholu můžeme dostat po hranách do všech zbývajících vrcholů, pak tento postup skončí tím, že nakonec budou všechny vrcholy probrané. Jestliže ale je graf nesouvislý a tedy některé vrcholy jsou z počátku nedosažitelné, pak po probrání jedné jeho komponenty zmizí všechny

dosažené vrcholy: všechny vrcholy, které byly dosaženy byly poté také probrány. Přitom ale vrcholy ostatních komponent grafu budou nedosažené. Proto jeden z nich v bodě 2 zvolíme a prohledávání pokračuje v jeCho komponentě. Na obrazovce je výchozí graf záměrně volen nesouvislý pro ilustraci prohledávání po komponentách, ale můžete jej přeeditovat na souvislý.

V této scéně je zobrazený graf neorientovaný a každou hranu je možno projít v libovolném směru. Pokud jsme ovšem v jednom směru prošli, hrana se označí jako probraná a procházet v opačném směru ji již algoritmus nemůže.

Prohledávání můžete provádět ve třech módech. Prohledávání je možno krokovat a nebo sledovat jako souvislou animaci; vrcholy a hrany volí Algovize náhodně. V ručním módu volí vrchol uživatel poklepáním a stejně tak v případě probíraného dosaženého vrcholu volí neprobranou hranu poklepáním; krok odporující výše uvedeným bodům 1 a 2 by Algovize neprovedla a místo toho by ukázala chybové hlášení.

Zkuste si nejprve prohledávání jako animaci a pak s krokováním. Na závěr ale si zkuste prohledávat grafy ručně tak dlouho, dokud několikrát za sebou neprovedete vše správně bez jakékoli chyby.

Scéna: *Strom prohledávání*

Předchozí scéna je zopakována ještě jednou, ale pozornost bude soustředěna na strukturu užitečných hran, tedy hran, pod kterými se při prohledávání objeví tlustá bílá šipka. Od předchozí scény se liší tím, že než začne vlastní prohledávání, graf se přesune do spodní části obrazovky, která ztmavne a během prohledávání jsou dosažené vrcholy přemisťovány do světlé horní části obrazovky a rozmisťovány tak, aby struktura užitečných hran byla dobře patrna.

Podívejte se, že z každého vrcholu v vychází *nejvýše jedna* tlustá bílá šipka, protože pokud bychom dvěma různými hranami prošli do vrcholu v, pak nejpozději po průchodu první z nich se vrchol v stane dosaženým a tedy druhá z těchto hran už bude klasifikována jako neužitečná.

Jistě si také povšimnete, že užitečné hrany vytvářejí strom nebo les (několik stromů), neboli graf bez cyklů. Je snadné dokázat to sporem. Předpokládejme, že v grafu užitečných hran je cyklus. Označme jako v ten z vrcholů cyklu, který byl dosažen jako poslední. Vrchol v má v cyklu dva sousedy; označme je u_1 a u_2. Když byl v dosažen, sousedé u_1 i u_2 už byly podle předpokladu dosažené nebo dokonce probrané. Kdybychom hranu spojující v a jeho souseda u_1 nebo u_2 procházeli vycházejíce z v směrem do souseda, museli bychom ji tedy označit za neužitečnou; proto by obě hrany spojující v s jeho sousedy musely být procházeny ve směru *do* vrcholu v. Užitečnost obou hran by ale byla ve sporu s tvrzením dokázaným v předchozím odstavci.

Vrcholy jsou v našem appletu po dosažení přemisťovány tak, aby strom prohledávání (tedy strom tvořený užitečnými hranami) byl kreslen způsobem,

který je v informatice obvyklý (m.j. tedy s kořenem nahoře a listy dole) a aby byl dobře přehledný. Neužitečné hrany jsou kresleny za užitečnými, aby nepřekážely pohledu na strom prohledávání. Povšimněte si, že užitečnými hranami jsme procházeli ve směru shora dolů, tedy v řeči stromu od již dosaženého vrcholu k jemu ještě nedosaženému synovi a tlusté bílé šipky jsou vlastně ukazateli na otce ve stromu.

Jelikož je obvykle obrazovka protažena do šířky, může se stát, že bude nákres ve vertikálním směru příliš stlačen. Pak jej můžete otočit o 90 stupňů doleva zatržením checkboxu **Horizontální.** Strom pak bude narůstat ve směru zleva doprava, ve kterém může být více prostoru pro přehledný nákres. Tato možnost bude k dispozici i v některých dalších scénách.

Scéna: *Obecné prohledávání orientovaného grafu*

V této scéně je prohledávaný graf orientovaný. Pravidla prohledávání jsou formulována stejně jako v minulé scéně, ale může se stát, že existuje hrana (v, w), ale nikoli (w, v), tedy je povolen přechod z vrcholu v do vrcholu w, ale nikoliv naopak. Zkuste si prohledávání znovu a uvidíte, že se orientovaný případ od neorientovaného příliš neliší.

Neorientovaný graf se často považuje za speciální případ orientovaného s tím, že neorientovaná hrana spojující dva vrcholy v a w se chápe jako dvojice opačně orientovaných hran (v, w) a (w, v). Takovýto "neorientovaný" graf se vytvoří, pokud zatrhnete volbu **Neorientovaný.**

V takovémto grafu ale prohledávání probíhá jinak než v předchozích scénách, protože projdeme-li hranu (v, w) a označíme-li ji jako probranou, opačná hrana (w, v) zůstává neprobranou a časem přes ni předeme také. Když ale hranou (w, v) budeme procházet, bude už vrchol v dávno dosažený nebo dokonce probraný a proto hrana (w, v) bude označena za neužitečnou a nic nového nepřinese. Faktický výsledek výpočtu je tedy vlastně stejný jako kdyby po průchodu hranou (v, w) jsme za probranou označili i hranu opačnou, což by odpovídalo tomu, jak se probírá neorientovaný graf.

Scéna: *Prohledávání grafu do šířky*

Obecné prohledávání je sice užitečný postup, například hledáme-li komponenty neorientovaného grafu, ale daleko důležitější jsou jeho speciální případy. V této scéně si probereme tak zvané prohledávání do šířky (BFS, z anglického Breadth-First Search).

Prohledávání do šířky je založeno na tom, že dosažené vrcholy se řadí do fronty a v bodě 1 algoritmu prohledávání se vždy volí ten dosažený vrchol, který je ve frontě první v pořadí, neboli ten, který je ve frontě nejdéle. Tento vrchol je tedy volen tak dlouho, dokud z něho vychází alespoň jedna neprobraná hrana. V okamžiku, kdy se všechny hrany z vrcholu vycházející

proberou, je překlasifikován na probraný a vyjmut z fronty a k probírání nastupuje ten, který stál za ním. Nově dosažené vrcholy se zařazují na konec fronty.

Jistě si také všimnete, že při prohledávání do šířky je každý probraný vrchol v libovolném okamžiku očíslován menším číslem než kterýkoli dosažený vrchol. Z dosažených vrcholů je vždy volen ten, který má nejmenší očíslování.

Fronta dosažených vrcholů je také schematicky znázorněna v dolní části obrazovky způsobem, který naznačuje postup vrcholů frontou. Klepnutí na obrázek vrcholu ve frontě ukáže referenci na obraz téhož vrcholu v grafu.

Prohledávání do šířky je možno opět provádět ve třech módech popsaných v předchozí scéně; v ručním módu je nutné dodržovat správný výběr vrcholu z fronty.

Scéna: *Prohledávání do šířky ještě jednou*

V této scéně ukážeme proč je prohledávání do šířky tak užitečné a často používané (viz například Dinitzův algoritmus pro toky v sítích).

Vrcholy budeme nyní označovat čísly *odlišně* od číslování v předchozí scéně; aby nedošlo k nedorozumění, budeme zde používat římské číslice. Počáteční vrchol (a obecně každý vrchol, který byl vybrán v bodu 2 algoritmu prohledávání přímo z množiny nedosažených vrcholů) bude nazýván *počátek* a bude označen nulou. Kdykoli se pak v bodě 2.(b).i přejde z nějakého vrcholu v do jiného vrcholu w, který zatím byl nedosažený, pak se vrchol w očísluje číslem o 1 větším, než je očíslování vrcholu v.

Očíslování vrcholů budeme také znázorňovat graficky stejně jako v druhé scéně této kapitoly, věnované stromu prohledávání: počátek se po očíslování přesune do horní části obrazovky. Jeho sousedi, což budou právě všechny vrcholy, které budou očíslovány I, se přesunou do vodorovné vrstvy, která leží kus pod počátkem. Vrcholy, do kterých se dostaneme poprvé z vrcholů první vrstvy, což jsou právě všechny vrcholy, které budou očíslovány II, budou v druhé vrstvě, ležící bezprostředně pod první vrstvou. Obecně vrcholy prvně dosažené z k-té vrstvy budou očíslovány číslem $(k+1)$ a přesunuty do $(k+1)$-ní vrstvy.

Jistě jste si všimli, že vrcholy jsou probírány po vrstvách; napřed počátek, pak vrcholy první vrstvy, pak druhé vrstvy atd. Není nutné, aby vrcholy v jedné vrstvě byly probírány systematicky zleva doprava, jak se to jeví v appletu. Toto pořadí je dáno implementovaným způsobem výběru hran vycházejících z probíraného dosaženého vrcholu. Zaškrtněte volbu **Hrany náhodně** a uvidíte prohledávání do šířky, kde sice jsou vrcholy probírány po vrstvách, ale pořadí v rámci vrstvy je zcela chaotické (ale pochopitelně jestliže je vrchol jednou zvolen, probereme všechny hrany z něj vycházející dříve než si vybereme další vrchol).

Pohledem na výsledné rozmístění vrcholů se stane jasnou hlavní vlastnost

prohledávání do šířky: očíslování libovolného vrcholu, tedy index vrstvy, ve které se na konci nachází, je rovno nejmenšímu počtu hran, které je nutno přejít při přesunu z počátku do uvažovaného vrcholu. Prohledávání do šířky je tedy algoritmus, řešící nejjednodušší variantu problému hledání nejkratší cesty v grafu, kdy délka cesty je dána počtem jejích hran. Ve stromu prohledávání do šířky totiž není žádná hrana, která by spojovala dvě vrstvy, které nejsou sousední, tedy která by jednu nebo více vrstev přeskakovala. Z nákresu to je dobře vidět, zkuste si rozmyslet, proč tomu tak je při prohledávání do šířky, ale nemusí být při obecném prohledávání.

Pohledem na výsledný graf také pro každý vrchol ihned zjistíte, jak se do něj z počátku dostat cestou s nejmenším možným počtem hran: je to cesta složená výhradně z užitečných hran, tedy z hran, které mají bílý podklad. Pro jednoduchost takové cestě budeme dále říkat *bílá* cesta. Je to cesta ve stromu prohledávání z kořene do daného vrcholu.

Bílou cestu popisované v předchozím odstavci nebudeme hledat ve směru z počátku (tedy kořene) do zvoleného vrcholu, protože není obecně jasné kterou z tlustých bílých šipek v protisměru přejít, abychom skončili ve zvoleném vrcholu. Je ale vidět, že je snadné ji nalézt pozpátku ze zvoleného vrcholu do počátku, protože z každého vrcholu s výjimkou počátku jde zpět jediná tlustá bílá zpětná šipka.

Hledání nejkratší cesty couváním je také důvod, proč jsou bílé podkladové šipky pod užitečnými hranami orientovány proti směru, kterým jsme hranou prošli.

Scéna: *Prohledávání do šířky v nesouvislém neorientovaném grafu*

Aniž to bylo výslovně uvedeno, Algovize nabízela v předchozí scéně souvislý graf. Pak jediným vrcholem, který se vybral přímo z nedosažených vrcholů v bodě 2 obecného schématu prohledávání do šířky byl vrchol, který byl zvolen na začátku výpočtu.

V této scéně je naopak volen graf, který je nesouvislý. Pak z prvního zvoleného vrcholu se dostaneme jen do některých vrcholů a když je komponenta zvoleného grafu probrána, dostaneme se do stavu, kdy neexistují dosažené (ale neprobrané) vrcholy a přitom prohledávání není ukončeno, protože vrcholy v další komponentě nebo komponentách jsou ještě nedosažené. Pak se v bodě 2 volí znovu počátek označený číslem 0 a dále se pracuje v jeho komponentě.

V orientovaném případě se připouští i grafy souvislé, které ale dovolují volit počátek, aby do některých vrcholů z něho nevedla žádná cesta.

Scéna je koncipována až na nesouvislost grafu podobně jako scéna předchozí a vlastně nepřináší nic nového než poznatek, že v nesouvislém grafu se pro každou komponentu opakuje prohledávání tak, jak bylo ukázáno v minulé scéně.

Scéna: *Prohledávání do hloubky*

Formálně je prohledávání do hloubky velmi podobné prohledávání do šířky: dosažené vrcholy se také řadí do fronty, ale na rozdíl od standardní fronty používané v předchozích dvou scénách, která bývá označována jako FIFO (first-in-first-out, neboli volen je vrchol, který do fronty přišel první a je tedy v ní nejdéle), prohledávání do hloubky používá frontu LIFO (last-in-first-out), kde je vždy vybrán vrchol, který do fronty přišel jako poslední a je tedy v ní nejkratší dobu. LIFO fronta bývá často nazývána *zásobník* a tento název budeme používat i zde, abychom vyloučili záměnu s FIFO frontou. Zásobník používaný při prohledávání do hloubky je ukázán v dolní části obrazovky tak, aby bylo patrno, že je k němu přístup jen z jedné strany.

Prohledávání do hloubky je pochopitelně možné provádět i v orientovaném grafu, ale v této scéně se omezíme na neorientované grafy, protože vlastnost, o které budeme mluvit v závěru scény a kterou budeme aplikovat při hledání artikulací grafu, platí v jednoduché podobě jen pro neorientované grafy.

Zkuste si krokovat prohledávání do hloubky a jistě zjistíte, že se podobá prohledávání bludiště, při kterém si označujeme již navštívená místa, kde se chodby setkávají (t.j. vrcholy grafu) a snažíme se postupovat stále dále. Zásobník dosažených vrcholů přitom popisuje, kudy vede červená nit, kterou si za sebou odvíjíme, abychom trefili zpátky do vstupu do bludiště. V appletu si cestu odpovídající posloupnosti vrcholů v zásobníku označujeme tak, že hrany, které do ní patří, mají červený podklad.

Za povšimnutí stojí, že prohledávání do hloubky se liší od prohledávání do šířky tím, že se všechny hrany vycházející z jednoho vrcholu neprohledávají najednou: jestliže z jistého vrcholu v vedou dvě neprobrané hrany a přechodem po první z nich se dostaneme do nedosaženého vrcholu w, pak začneme hned probírat vrchol w a druhou hranu vycházející z v odložíme na později.

Scéna: *Strom prohledávání do hloubky*

Předchozí scénu zopakujeme s tím, že dosažené vrcholy budeme přemisťovat tak, aby byl dobře uspořádán výsledný strom prohledávání. Vrcholy grafu prohledávaného do hloubky budeme také číslovat v pořadí, v jakém byly dosahovány. Toto číslování nemá na první pohled žádnou bezprostřední aplikaci podobnou rozdělení do vrstev při prohledávání do šířky. Ukážeme ale, že prohledávání do hloubky má jednu zajímavou vlastnost, která se sice může zdát na první pohled k ničemu, ale je na ní možno založit několik velmi rychlých algoritmů pro řešení různých problémů teorie grafů, z nichž jeden uvidíme v posledních třech scénách této kapitoly.

Proveďte prohledávání až do konce a pak se podívejte na strom užitečných hran (tedy hran s bílým podkladem). Zvolte si libovolnou neužitečnou hranu a nazvěte její konce jako v a w. Pak buď v leží na (jednoznačně určené) bílé cestě složené z užitečných hran z w do počátku nebo naopak w leží na

(jednoznačně určené) bílé cestě z v do počátku. Jinými slovy, jeden z konců libovolné neužitečné cesty musí být předchůdcem druhého z konců ve stromu užitečných hran. Znamená to, že neužitečná hrana nemůže být příčkou mezi dvěma různými větvemi bílého stromu.

Scéna: *Komponenty grafu*

V této scéně si ukážeme, jak se hledají komponenty grafu. Přitom se omezíme na neorientované grafy; pojem komponenty je také možno definovat pro orientované grafy, ale je komplikovanější a komponenty orientovaného grafu se také obtížněji hledají a proto se jimi v této knize zabývat nebudeme.

Řekneme, že dva vrcholy neorientovaného grafu patří do stejné komponenty, pokud se z jednoho z nich po hranách dostaneme do druhého. Množina vrcholů se tedy rozpadne do navzájem disjunktních podmnožin, které nazýváme komponenty. Pro čtenáře vzdělaného v diskrétní matematice uvádím, že relace "být ve stejné komponentě" je ekvivalence (je reflexivní, symetrická a tranzitivní) a komponenty jsou ekvivalenční třídy této ekvivalence. Ostatní ať se podívají na obrazovku a jistě komponenty zobrazeného grafu jasně uvidí (Algovize vytváří graf záměrně tak, aby komponenty nebyly zapleteny do sebe).

Komponenty vlastně již umíme určovat: prohledávejte graf libovolným způsobem popsaným výše v této kapitole, tedy postupem podle obecného algoritmu z první scény a nebo třeba prohledáváním do šířky nebo do hloubky. Když vybereme vrchol v bodě 2 přímo z nedosažených vrcholů, začínáme novou komponentu. Všechny vrcholy, které jsou poté označovány za dosažené v bodě 1.(b).i, patří do stejné komponenty a v okamžiku, kdy zjistíme, že v grafu nejsou žádné dosažené vrcholy, probírání komponenty skončilo (a pokud není probrán celý graf, začne vzápětí zpracovávání další komponenty).

Zkuste si prohledávání, v této scéně nejsou probrané vrcholy černé, ale mají tmavé barvy volené tak, že vrcholy každé komponenty jsou obarveny stejně, ale různé komponenty používají různé barvy. Přepnutí barev pro probrané vrcholy se tedy provádí při každém vstupu do bodu 2 algoritmu.

Scéna: *Artikulace grafu*

V této scéně se budeme zabývat opět jenom neorientovanými grafy. Na rozdíl od předchozí scény, kde by použití souvislých grafů neukazovalo dostatečně názorně použití prohledávání při hledání komponent, se zde omezíme na grafy, které jsou sice souvislé, ale své souvislosti mohou snadno pozbýt.

Artikulace v souvislém grafu je vrchol, jehož vynecháním (spolu s hranami, které z tohoto vrcholu vycházejí) se graf stane nesouvislým. Jinými slovy, graf se po vytržení vrcholu rozpadne na alespoň dva kusy, které nejsou žádným způsobem propojeny. Ještě jinak je vrchol u artikulace, jestliže existují dva vrcholy v a w takové, že každá cesta z v do w musí procházet přes u.

V grafu na obrazovce jsou artikulace označeny červeným podkladem pod obrázkem vrcholu. Cílem této scény je pouze pojem artikulace osvětlit; jejich hledáním se budeme zabývat v příští scéně.

Pojem artikulace se také dá definovat na základě 2-souvislých komponent. Řekneme, že množina A vrcholů je *2-souvislá*, jestliže pro každé dva různé vrcholy $v, w \in A$ existují alespoň *dvě* cesty spojující v a w, které mají společné pouze tyto koncové vrcholy. Řekneme, že množina vrcholů je *2-souvislá komponenta*, pokud je 2-souvislá a není vlastní částí jiné 2-souvislé množiny.

Zvolte si na ovladači **Ukaž 2-komponenty** a klepněte na některý vrchol, který není artikulací. Vrcholy 2-souvislé komponenty, která obsahuje zvolený vrchol, se zbarví růžově. Povšimněte si, že součástí komponenty obecně jsou i artikulace. (Klepnutí mimo vrchol vrátí zvolené 2-souvislé komponentě její původní barvu).

Nyní si zvolte na ovladači **Ukaž cesty** a klepněte do dvou různých vrcholů zvolené 2-souvislé komponenty. Objeví se dvě cesty, které zvolené vrcholy propojují a s výjimkou konců nemají společné vrcholy. Pohrajte si s volbou 2-souvislých komponent a propojujících cest, aby vám bylo dostatečně jasné, co to 2-souvislé komponenty jsou. (Pokud zvolené vrcholy nejsou ve stejné 2-souvislé komponentě, neobjeví se cesty, ale chybové hlášení).

Patrně jste si všimli, že každý vrchol se nachází alespoň v jedné 2-souvislé komponentě a především, že artikulace jsou *právě* ty vrcholy, které leží ve více komponentách. Jinými slovy, artikulace jsou průniky 2-souvislých komponent. (To také vysvětluje, proč když jste klepli při ukazování 2-souvislých komponent na artikulaci, nic se nestalo - ne proto, že by nebyla obsažena v žádné 2-souvislé komponentě, ale přesně naopak: leží ve více 2-souvislých komponentách najednou a není jasné, která z nich by měla být zvýrazněna a jiná ne).

Scéna: *Hledání artikulací*

V minulé scéně jsem vysvětlil, co to je artikulace v neorientovaném grafu. Zde si ukážeme patrně nejrychlejší algoritmus pro hledání artikulací.

Popisovaný algoritmus je prohledávání grafu do hloubky, doplněné o získávání dalších údajů, na základě kterých každému vrcholu v přiřadíme dvě čísla $P(v)$ a $LOWPT(v)$. První je pořadové číslo vrcholu; jestliže vrchol v byl dosažen jako k-tý, bude $P(v) = k$. Druhé číslo bude vysvětleno později. Před začátkem prohledávání je třeba manuálně označit vrchol, ze kterého se bude prohledávat (tedy který bude jako první zvolen za dosažený).

Prohledejte strom do hloubky buď po krocích knoflíkem **Krok** nebo najednou knoflíkem **Prohledej.** Pak se podívejte podrobněji na strom prohledávání. Jelikož kořen se chová odlišně od ostatních vrcholů, probereme jeho vlastnosti později, nyní klepněte na některý jiný vrchol v stromu. Zvolený vrchol v zčervená, jeho následníci ve stromu prohledávání zrůžoví a vrcholy

cesty z kořene do zvoleného vrcholu ve stromu prohledávání zmodrají.

Barva užitečných hran incidentních se zvoleným vrcholem se změní a bude velmi blízká barvě pozadí, takže je dobře vidět, co by zbylo ze stromu prohledávání, kdybychom vytrhli zvolený vrchol v. Komponenty tohoto stromu by byly jednak podstromy zakořeněné v synech zvoleného vrcholu (které jsou růžové) a pak zbytek stromu, který zahrnuje modře zbarvené vrcholy cesty z kořene k předchůdci zvoleného vrcholu v. Aby vrchol v *nebyl* artikulací, musí být tyto komponenty propojeny do souvislého celku neužitečnými hranami. Aby bylo dobře vidět, zda takové propojení existuje, zbarví se neužitečné hrany, které propojují různé komponenty stromu s vytrženým v zeleně a ostatní neužitečné hrany (tedy hrany uvnitř komponent) takřka splynou s pozadím.

Z vlastnosti prohledávání do hloubky, které bylo uvedeno v příslušné scéně, plyne, že modré vrcholy jsou navzájem propojeny užitečnými hranami a mezi souvislými podstromy dvou různých synů zvoleného vrcholu v nemůže existovat žádná hrana. Jinými slovy, jakákoli hrana, která z takových podstromů vede ven, musí vést do v nebo do některého z modrých vrcholů. Vzájemné propojení komponent stromu prohledávání s vytrženým v, o kterých jsme mluvili v předchozím odstavci, je tedy možné jen tak, že *každý* z podstromů synů vrcholu v je alespoň jednou zelenou hranou spojen s některým modrým vrcholem.

Tohoto pozorování využijeme následujícím způsobem: pro každý vrchol w označíme jako $LOWPT(w)$ je minimum z následujících čísel:
$P(w)$, a $P(z)$ pro každý vrchol z, který je hranou spojen s w nebo s některým jeho následníkem ve stromu prohledávání.

Výhodný způsob výpočtu funkce $LOWPT$ bude ukázán v následující scéně, nyní se podíváme, jak ji využít pro určení, zda je daný vrchol v artikulací (předpokládám, že v je zvolený červený vrchol na obrazovce).

Podstrom, určený synem w vrcholu v ve stromu prohledávání je spojen hranou s modrým vrcholem právě když $LOWPT(w) < P(v)$: pořadová čísla růžových následníků vrcholu v jsou větší než $P(v)$, zatímco pořadová čísla modrých vrcholů cesty z kořene do v jsou menší než $P(v)$. Jestliže některý vrchol x podstromu syna w vrcholu v je spojen s modrým vrcholem z, pak podle definice je $LOWPT(w)$ nejvýše rovné $P(z)$, tedy méně než $P(v)$. Naopak, pokud je $LOWPT(w) < P(v)$, pak některý následník vrcholu w, tedy prvek jeho podstromu, musí být spojen s vrcholem z, pro který $P(z) < P(v)$, což je možné jen pro modré vrcholy.

Odtud již plyne následující kritérium: nekořenový vrchol v je artikulací právě když má syna w ve stromu prohledávání do hloubky, pro kterého je $P(v) \leq LOWPT(w)$.

Zatrhněte nyní volbu **Ukaž LOWPT.** Vrcholy stromu se rozšíří a bude v nich ukázáno obojí očíslování ve tvaru $P(v) : LOWPT(v)$. Volte si různé

vrcholy v a sledujte platnost právě uvedeného kritéria i jeho souvislost s tím, jak jsou podstromy synů zvoleného vrcholu napojeny nebo nenapojeny před vrchol v do modré cesty.

Nakonec probereme, jak určit, zda je artikulací kořen. V jeho případě neexistují modré vrcholy na cestě z kořene do jeho předchůdce a proto podstromy jeho synů není možno nijak mezi sebou propojit, neboť jejich propojení jinak než přes kořen by nutně potřebovalo příčku mezi podstromy, která ve stromu prohledávání do hloubky neexistuje.

U kořene je proto kritérium velmi jednoduché: kořen stromu prohledávání je artikulací uvažovaného grafu právě tehdy, když ve stromu prohledávání do hloubky má alespoň dva syny.

Scéna: *Výpočet $LOWPT$*

Tato scéna ukáže výpočet funkce $LOWPT$, která byla definována a použita v předchozí scéně. Je snadné ověřit, že $LOWPT(v)$ je minimum z následujících čísel:
$P(v)$, $P(z)$, kde (v, z) je hrana grafu, a $LOWPT(w)$, kde w je syn vrcholu v ve stromu prohledávání.

Je to z toho důvodu, že hrany z následníků vrcholu v ve stromu prohledávání, o kterých se v definici $LOWPT(v)$ v minulé scéně mluví, jsou zahrnuty v hodnotách funkce pro syny vrcholu v.

Algoritmus prohledávání do hloubky proto stačí doplnit pro výpočet funkce $LOWPT$ takto:

1. pokud byl vrchol v vybrán jako dosažený přímo v bodě 2 algoritmu prohledávání (viz první scéna této kapitoly),
   položíme $LOWPT(v) = P(v)$;

2. pokud byl vrchol w překlasifikován na dosažený po průchodu hranou (v, w) v bodě 1.(b).i algoritmu prohledávání,
   položíme $LOWPT(w) = P(v)$;

3. pokud byla hrana (v, w) probrána a překlasifikována na neužitečnou v bodě 1.(b).ii algoritmu prohledávání,
   položíme $LOWPT(v) = \min(LOWPT(v), P(w))$;

4. pokud byl vrchol w označen za probraný v bodě 1.(a) algoritmu prohledávání a pokud není počátkem prohledávání a v je jeho otcem ve stromu prohledávání,
   pak položíme $LOWPT(v) = \min(LOWPT(v), LOWPT(w))$.

Při prohledávání se v této scéně zobrazuje číslo použitého pravidla a červeným podkladem je vždy označen vrchol, jehož $LOWPT$ se změnilo.

Při aplikaci pravidla 3. hrana (v, w) dosahuje k některému z předchůdců vrcholu v ve stromu prohledávání a proto může snížit $LOWPT(v)$ na hodnotu $P(w)$.

Poznamenejme ještě, že vrchol v, který je otcem vrcholu w v 4. pravidlu, je vrchol do kterého vede z w tlustá bílá šipka na obrázku a zároveň ten vrchol, který se objeví na vrcholu zásobníku dosažených (ale neprobraných) vrcholů prohledávání do hloubky, když se vrchol w stane v bodě 1.(a) probraným a tedy je ze zásobníku odebrán.

Zkoušejte si prohledávání různých grafů z různých počátečních vrcholů (počáteční vrchol je třeba na začátku zvolit) a sledujte obě očíslování vrcholů. Jejich výchozí hodnota je nastavena jakmile se vrchol stane dosaženým (pro nedosažené vrcholy nejsou definovány) a hodnota P už se dále nemění, kdežto hodnota $LOWPT$ se ještě může zmenšovat. Při krokování se navíc objeví kroky úpravy $LOWPT$ při probrání neužitečné hrany nebo po vyjmutí probraného vrcholu ze zásobníku. Při nich je zvýrazněno pozadí hodnoty $LOWPT$, která by se mohla změnit a zvýrazněna buď probíraná neužitečná hrana nebo hrana od probraného vrcholu k jeho předchůdci ve stromu i zásobníku.

Kapitola 14

Extremální cesty v grafu

Tento applet ukazuje tři algoritmy pro hledání nejkratší nebo nejlacinější cesty v orientovaném grafu. Je dán orientovaný graf spolu s cenami hran. Ceny jsou čísla přiřazená hranám a můžeme se na ně dívat jako na poplatek, který je nutno zaplatit za projetí hranou. Cena může být i záporná - z blíže neurčených důvodů může cestovatel za projetí dostat zaplaceno místo aby platil. V případě, kdy ceny hran musí být nezáporné (např. u Dijkstrova algoritmu), používáme obvykle místo výrazu "cena" označení "délka" hrany a mluvíme o nejkratší cestě. Cena nebo délka cesty je dána součtem cen nebo délek jejích hran.

Dále je dány dva vrcholy v a w grafu a naším úkolem je nalézt nejlacinější cestu z v do w.

V praxi je takto formulovaná úloha (nalezení nejlacinější cesty mezi dvěma zvolenými vrcholy) nejčastější, ale všechny dosud známé algoritmy ji řeší tak, že hledají nejlacinější cestu z jednoho vrcholu do všech (nebo duálně ze všech vrcholů do jednoho). Problém "z jednoho do jednoho" se od problému "z jednoho do všech" liší jen tím, že hledání z jednoho do jednoho lze někdy ukončit dříve - když je nalezena prokazatelně nejkratší cesta do cílového vrcholu.

Jak se ukazuje, potíž s nejlacinější cestou mezi dvěma vrcholy u a v může nastat v případě, kdy některá cesta se dotýká cyklu, který má záporný součet cen hran. V takovém případě neexistuje nejlacinější cesta z u do v, protože je možno z u dojít do některého vrcholu záporného cyklu, ten pak mnohokrát oběhnout a pak teprve dojít do v. Čím víckrát se cesta oběhne, tím je "lacinější" je cesta (její cena je zápornější) a minimum z cen takových cest neexistuje.

Pokud by nás zajímala nejlacinější *prostá* cesta, dostali bychom úlohu, která je sice jednoznačně a korektně definovaná, ale NP-těžká. Je mimo rámec této knihy rozebírat teorii NP-úplnosti, ale zhruba řečeno to znamená, že nejsou známy dostatečně rychlé algoritmy, které by ji řešily (a možná takové

algoritmy ani neexistují). Proto je třeba nějakým způsobem omezit zadání úlohy hledání nejlacinější cesty mezi danými dvěma vrcholy. Každý z našich tří algoritmů bude předpokládat omezení jiné síly a lze říci, že čím silnější omezení, tím jednodušší a rychlejší algoritmus můžeme najít.

První algoritmus, který budeme nazývat algoritmus kritické cesty, protože je základem pro známou metodu kritické cesty (CPM - critical path method) a PERT (program evaluation and review technique), vyžaduje, aby graf byl acyklický, to znamená, aby v něm neexistoval vůbec žádný orientovaný cyklus. Výměnou za velmi silné omezení vstupních dat dostaneme velmi jednoduchý a rychlý algoritmus (pracuje v čase úměrném součtu počtu vrcholů a hran grafu). Pokud je algoritmu dán graf s cyklem, pak to pozná a může být upraven tak, aby v takovém případě (některý) cyklus popsal.

Druhý algoritmus je Dijkstrův algoritmus z roku 1959, který vyžaduje, aby ceny hran byly nezáporné, ale připouští cykly. Vzhledem k tomu, že v mnoha aplikacích hrany bývají fyzicky existující spoje, například silnice, železnice nebo elektrická vedení mezi městy a ohodnocení hran obvykle souvisejí s jejich délkou, není požadavek nezápornosti cen hran obvykle omezující.

Výhodou Dijkstrova algoritmu je jeho jednoduchost a rychlost. Primitivní implementace pracuje v čase $O(n^2)$, zatímco při použití pokročilých datových struktur (např. Fibonacciova halda) pracuje v čase $O(n \log n + m)$, kde n je počet vrcholů a m je počet hran grafu. Poznamenejme, že platnost vstupní podmínky nebo případný důvod její neplatnosti je možno ověřit velmi snadno probráním hran.

Nejobecnější vstupní podmínku z algoritmů zde uvedených má Bellman-Fordův algoritmus, vycházející ze dvou původních prací Bellmana (1958) a Forda (1962). Tomu stačí, aby v grafu neexistoval cyklus se záporným součtem cen hran, který je dosažitelný z výchozího vrcholu. Jak bylo uvedeno výše, pokud by takový cyklus existoval, není dobře možné nadefinovat, co to je nejkratší vzdálenost mezi dvěma vrcholy, takže lze říci, že Bellman-Fordův algoritmus je možné použít vždy, kdy zadání dává smysl.

Bellman-Fordův algoritmus je také poměrně jednoduchý, ale za obecnost vstupní podmínky platíme rychlostí výpočtu. Odhad pro dobu výpočtu v nejhorším případě je $O(n(n+m))$, kde n je počet vrcholů a m je počet hran grafu, což je podstatně horší odhad než v předchozích dvou případech a proto je Bellman-Fordův algoritmus ve většině případů méně výhodný než Dijkstrův algoritmus. Má ale další, původně asi nezamýšlenou oblast použití: detekci záporného cyklu v grafu, což je úloha, která se využívá například při hledání nejlacinějšího maximálního toku v síti.

14.1 Algoritmus kritické cesty

Scéna: *Topologické uspořádání acyklického grafu*

Základem pro metodu kritické cesty je t. zv. topologické uspořádání acyklického orientovaného grafu. Je to uspořádání vrcholů grafu do posloupnosti $v_1, \ldots, v_n$ tak, že je-li (v_i, v_j) orientovaná hrana grafu, pak $i < j$. Hrany tedy jdou výhradně "zleva doprava".

Je zřejmé, že pokud je v grafu orientovaný cyklus, topologické uspořádání vrcholů nemůže existovat. Toto tvrzení platí i naopak: pokud v orientovaném grafu není orientovaný cyklus, pak existuje topologické uspořádání jeho vrcholů. Zde si to ukážeme pro konečné grafy.

Pokud je dán orientovaný konečný graf bez cyklu, pak následující postup topologické uspořádání vždy najde:

Vrcholy označíme červenou barvou a potom dokud alespoň jeden vrchol zůstává červený, opakujeme následující programový cyklus:
zvolíme libovolně červený vrchol, do kterého nevede žádná hrana začínající v jiném červeném vrcholu, a obarvíme jej na zeleno.

Jestliže se vrcholy očíslují v pořadí, v jakém se stávaly zelenými, dostaneme zcela zjevně topologické uspořádání vrcholů.

Na začátku i po každém stisknutí knoflíku **Nový graf** vám Algovize předloží acyklický graf. Postupně klikejte na červené vrcholy, do kterých nevede hrana z jiného červeného vrcholu. Kliknutý vrchol zezelená a přemístí se do horní části obrazovky, kde se nakonec vrcholy seřadí v topologickém uspořádání. Algovize vás také upozorní, kliknete-li na špatný vrchol.

Scéna: *Vrchol nulového stupně v acyklickém grafu*

Postup v minulé scéně je korektně definován pouze v případě, že pro libovolnou množinu červených vrcholů v konečném acyklickém orientovaném grafu existuje alespoň jeden její vrchol, do kterého nevede žádná hrana z červeného vrcholu. V minulé scéně jste jistě takový vrchol vždy našli; nyní ukážeme, že tomu tak vždy musí být.

Předpokládejme opak a uložme značku do libovolného červeného vrcholu. Do vrcholu se značkou vede alespoň jedna hrana z červeného vrcholu. Jednu z nich vybereme a značku přesuneme proti směru hrany. Tím se dostaneme zase do červeného vrcholu. Posun proti směru hrany do jiného červeného vrcholu můžeme opakovat a tento pohyb může trvat do nekonečna. Jelikož ale je graf konečný, dostali bychom se po jisté době do vrcholu, kde jsme již byli, což by znamenalo, že jsme v protisměru oběhli cyklus v grafu - to by ale byl spor s předpokladem, že je graf acyklický.

V konečném acyklickém orientovaném grafu se tedy pohybem proti směru hran v rámci množiny červených vrcholů musíme dostat do vrcholu, ze kterého není pokračování, to znamená do vrcholu do kterého nevede červená hrana.

Zkuste si generovat náhodné grafy knoflíkem **Nový graf** nebo je vytvářet editorem, poté klepnutím na některý vrchol do něho vložit značku a pak knoflíkem **Krok** couvat v grafu nebo knoflíkem **Animace** couvání spustit jako souvislou akci. Couvat je možno i manuálně, kliknutím na příslušnou hranu. Hrany, přes které se couvalo se barevně odliší. V závislosti na tom, jaký byl vytvořen graf, couvání značky vede buď k nalezení vrcholu, do kterého nevede žádná hrana, nebo naopak k nalezení cyklu. V obou případech je výsledek zvýrazněn a oznámen.

Je tedy zřejmé, že v acyklickém grafu tedy vždy nalezneme vrchol bez vstupní hrany. Jelikož podgraf acyklického grafu je zase acyklický, dojde ke stejnému výsledku i po odtržení libovolného počtu vrcholů z původního grafu. Celkově tedy vidíme, že je možno graf topologicky uspořádat *právě když* je acyklický. Z výše uvedeného vyplývá i nepříliš rychlý algoritmus, který topologické uspořádání najde nebo prokáže, že neexistuje.

Scéna: *Rychlé určení topologického uspořádání*

Určování červeného vrcholu, do kterého nevede hrana z žádného jiného červeného vrcholu, bylo v předchozí scéně prováděno způsobem, který sice poskytl existenční důkaz topologického uspořádání, ale pro svoji pomalost není vhodný pro praktický výpočet. Když již ale víme, že v konečném acyklickém grafu takový vrchol musí vždy existovat, lze algoritmus provádět daleko lépe.

U každého červeného vrcholu si budeme poznamenávat, kolik hran do něho vede z červených vrcholů (a pro vrchol v budeme tento počet označovat jako $D(v)$). Dále si budeme pamatovat množinu Q červených vrcholů v, pro které je $D(v) = 0$.

Algoritmus pracuje následujícím způsobem (předpokládáme, že všechny vrcholy jsou na začátku červené):

Krok: *Nulování pole D a množiny Q*

Pro každý vrchol v se položí $D(v) = 0$ a položí se $Q = \emptyset$. $\diamond$

Krok: *Určení stupňů*

Pro každou hranu (u, v) grafu se $D(v)$ zvýší o 1. $\diamond$

Krok: *Inicializace množiny Q*

Pro každý vrchol v se ověří, zda $D(v) = 0$ a pokud tomu tak je, pak se v vloží do Q. $\diamond$

Krok: *Cyklus*

Dokud je množina Q neprázdná, provádí se následující akce:
vybere se libovolný vrchol v z množiny Q, přebarví se na zeleno a pro každou hranu (v, w) vycházející ze zvoleného vrcholu v se $D(w)$ sníží o 1 a pokud tím klesne na nulu, zařadí se w do množiny Q. $\diamond$

Uvedený postup je na obrazovce ilustrován takto:
Vrcholy, které patří do množiny Q jsou zobrazeny na bílém podkladě. Hodnota $D(v)$ je udána číselně vedle vrcholu v. Vrcholy přebarvené na zeleno se řadí podobně jako v minulé scéně.

Výpočet je možno provádět v krocích, skocích nebo souvislou animací. Skoky provádějí operace nulování, určení stupňů a inicializace Q najednou, kroky se provádějí po jednotlivých vrcholech, resp. hranách. Dále se skokem provedou všechny akce spojené s obarvením jednoho vrcholu na zeleno najednou, kroky nejprve přebarví a přesunou vrchol a pak probírají po jedné z něho vycházející hrany.

Scéna: *Algoritmus kritické cesty*

Jakmile je topologické uspořádání vrcholů acyklického orientovaného grafu určeno, je velmi snadné nalézt ceny nejlacinějších cest z daného výchozího vrcholu v_0 do všech ostatních vrcholů grafu.

Každému vrcholu v bude přiřazena proměnná $L(v)$. Nejprve vrcholu v_0 přiřadíme hodnotu 0 a všem ostatním vrcholům hodnotu ∞ (pokud u vrcholu v zůstane až do konce výpočtu, bude to znamenat, že do vrcholu nevede z v_0 žádná orientovaná cesta; v průběhu výpočtu bude znamenat, že žádná taková cesta dosud nebyla nalezena).

Potom procházíme vrcholy v pořadí daném topologickým uspořádáním a každému vrcholu v přiřadíme minimum přes všechny hrany $h = (w, v)$ vedoucí do v z hodnot $L(w) + W(h)$, kde $W(h)$ je cena hrany h.

Pokud je pro některou hranu (w, v) hodnota $L(w)$ rovna ∞, pak pochopitelně také $L(w) + W(h) = \infty$ (pokud se z v_0 nedostanu do w, pak se také z v_0 přes w nedostanu do v). Jestliže do vrcholu v nevede hrana, pak zůstane $L(v)$ rovno ∞ (je na to možno se také dívat tak, že minimum přes prázdnou množinu hran je ∞, což je běžná matematická konvence).

Povšimněte si, že pokud (w, v) je hrana, pak w předchází v v topologickém uspořádání a proto v okamžiku výpočtu $L(v)$ je hodnota $L(w)$ již určena. Toto je přesně důvod, proč při výpočtu potřebujeme topologické uspořádání vrcholů.

Projděte si nyní celý výpočet; v závislosti na volbě na ovladači se buďto výpočet topologického uspořádání krokuje jako bylo uvedeno výše nebo se provede najednou.

14.2 Dijkstrův algoritmus

Scéna: *Dijkstrův algoritmus staticky*

Dijkstrův algoritmus připouští v grafu cykly, ale na druhé straně na rozdíl od algoritmu kritické cesty dovoluje jen hrany s nezápornou cenou. Místo "cena" budeme v této subkapitole říkat "délka". Délku hrany $h = (u, v)$ budeme označovat $\ell(h)$ nebo $\ell(u, v)$.

Pravidelně nejčastější aplikací algoritmů pro hledání nejkratší cesty je situace, kdy hrany grafu jsou silnice nebo železnice spojující města nebo jiné fyzické oblasti a cena hrany je dána její fyzickou délkou. V takových případech jsou délky hran kladné a proto uvedené omezení není v praxi většinou závažné.

Dijkstrův algoritmus je v zásadě speciálním případem prohledávání grafu a proto zde budeme používat terminologie, zavedené pro prohledávání. Vrcholy budou klasifikovány jako nedosažené, dosažené a probrané. Dělení hran na probrané a neprobrané zde sice není příliš potřeba, ale budeme se ho také držet. Často ale budeme používat dělení probraných hran na užitečné a neužitečné.

Na rozdíl od prostého prohledávání zde ještě budeme pro každý vrchol v používat proměnnou $E(v)$ (z anglického *E*stimation, tedy odhad), která bude představovat horní odhad vzdálenosti z počátku do vrcholu v a na konci výpočtu se této vzdálenosti bude rovnat. Zhruba řečeno, $E(v)$ udává délku *zatím* nejkratší známé cesty do vrcholu v; co to znamená "zatím nejkratší známá" cesta vysvětlím později.

Dijkstrův algoritmus používá následující strategii výběru dosaženého vrcholu jako výchozího vrcholu pro procházení hranami: jako aktivní se vybere dosažený vrchol v *minimalizující* hodnotu $E(v)$ mezi dosaženými vrcholy a proberou se všechny hrany z tohoto vrcholu vycházející. Hodnoty proměnné E pro některé dosažené vrcholy se při tom mohou měnit a některým dříve nedosaženým vrcholům se při jejich dosažení nastaví počáteční hodnota proměnné E, ale uvidíme, že tyto hodnoty nebudou nikdy menší než $E(v)$ zvoleného vrcholu v.

Probrání hrany (v, w) vlastně znamená, že jsme prozkoumali novou cestu nebo cesty do w, které využívají dosud známou cestu nebo cesty do u prodloužené o hranu (v, w) tak, aby končila(y) až ve w. Jak jsme řekli, nejkratší "dosud známá" cesta do v má délku $E(v)$ a tedy její prodloužení o hranu (v, w) znamená, že také známe novou cestu do w, která má délku $E(v) + \ell(v, w)$. Pokud je současná hodnota $E(w)$ větší než toto číslo, pak jej na délku právě objevené cesty přes v snížím.

Algoritmus tedy pracuje takto:

počátek v_0 se označí jako dosažený a položí se $E(v_0) = 0$;
ostatní vrcholy se označí jako nedosažené;
pak dokud existuje alespoň jeden dosažený (ale neprobraný) vrchol, opakuje se následující činnost:
jako v se označí dosažený vrchol, který má nejmenší hodnotu $E(v)$;
pro každou hranu (v, w) vycházející z v se provede následující:
jestliže v je nedosažený, pak se překlasifikuje na dosažený
a položí se $E(w) \leftarrow E(v) + \ell(v, w)$;
jinak pokud $E(w) > E(v) + \ell(v, w)$,
tak se položí $E(w) \leftarrow E(v) + \ell(v, w)$;
po probrání všech hran vycházejících z v se v označí za probraný.

Vybraný vrchol v, který se probírá v cyklu algoritmu, budeme pro zjednodušení výkladu nazývat *aktivní* vrchol.

Zkuste si výpočet algoritmu pro graf na obrazovce nebo graf, který si sami vytvoříte nebo zvolíte z nabídky příkladů. Poznamenávám, že se držíme barevné klasifikace vrcholů a hran, zavedené v kapitole o prohledávání: nedosažený vrchol žlutý, dosažený zelený, probraný vrchol černý, neprobraná hrana modrá a probraná černá. Výjimky z tohoto pravidla budou počátek (na začátku výpočtu modrý), aktivní vrchol v a probíraná hrana budou červené a konec w probírané hrany vycházející z červeného v bude růžový. U každé hrany je poznamenána její délka, u probraných a dosažených vrcholů je uvedena hodnota $E(v)$.

Pod některými hranami se také objevují zpětné bílé šipky, ukazující pro konec hrany, odkud jsme se do něho dostali. Jestliže je probírána hrana (v, w) vedoucí ze zvoleného dosaženého vrcholu v do *nedosaženého* vrcholu w, objeví se po probrání bílá šipka vedoucí z w zpět do v a ukazující, odkud jsme se do w poprvé dostali.

Proti kapitole o prohledávání zde je ale jistá změna: pokud je při probírání hrany (v, w) vrchol w dosažený, ale dojde ke změně $E(w)$ (tedy platilo $E(w) > E(v)+\ell(v, w)$), pak bílá šipka, která již z vrcholu w někam směřovala, se *přesměruje* do vrcholu v. Bílá šipka tedy neukazuje, odkud jsme se do vrcholu w dostali poprvé, ale odkud jsme se do něho dostali nejlépe (nejkratším způsobem).

Na začátku výpočtu můžete klepnutím na vrchol zvolit jiný počátek nebo o to požádat Algovizi. Zkuste si výpočet algoritmu projít; je možno nechat zobrazit zjednodušený program, ve kterém je zvýrazněn právě prováděný příkaz.

V názvu scény je slovo “staticky”. Netýká se vlastního algoritmu nebo jeho výpočtu, ale toho, jak výpočet ukazujeme na obrazovce. V této scéně se polohy vrcholů grafu nemění (proto statický) a průběh výpočtu je znázorňován měnícími se barvami vrcholů a hran. Výpočet proto není příliš názorný, takže krokováním algoritmu v této scéně nemusíte strávit příliš mnoho času.

V následující scéně si vše zopakujeme, ale graf překreslíme tak, aby bylo lépe patrno, co se při výpočtu děje.

Scéna: *Dijkstrův algoritmus*

V této scéně se výpočet ze scény předchozí zopakuje ještě jednou, ale (podobně jako jsme to dělali u prohledávání) vrcholy budeme přesouvat tak, aby výpočet byl názornější; pochopitelně se zachováním propojení vrcholů i všech ohodnocení vrcholů a hran. Jelikož je ale šířka okna na obrazovce obvykle větší než jeho výška, nebudeme vznikající strom kreslit obvyklým způsobem shora dolů, ale zleva doprava, protože tak získáme více prostoru pro pohyb vrcholů.

Než zahájíme vlastní výpočet, přesune se zvolený počátek do levé části obrazovky do tmavější oblasti, určené pro dosažené vrcholy a ostatní vrcholy se natlačí doprava do světlé oblasti, ve které budou nedosažené vrcholy. Dosažené vrcholy budou v tmavější oblasti uloženy a přesouvány tak, že jejich y-ová souřadnice (výška) bude zůstávat stále stejná, ale x-ová souřadnice vrcholu v (nebo přesněji rozdíl x-ové souřadnice vrcholu v a x-ové souřadnice počátku) bude úměrná hodnotě $E(v)$.

U vrcholů, pro které je $E(v)$ definováno, je uvedena dvojice čísel $E(v) : D(v)$, kde $D(v)$ je vzdálenost vrcholu v od počátku. Hodnota $D(v)$ je číslo, které máme nakonec vypočítat a uvádím jej proto, abyste proti němu mohli sledovat, jak výpočet postupuje.

Jestliže je probírán vrchol v a z něho vystupující hrana (v, w), pak pokud byl vrchol w nedosažený a tedy ležel ve světlé oblasti napravo, je vrcholem v "přitažen" do tmavější oblasti dosažených vrcholů a to do místa určeného odhadem $E(w) = E(v) + \ell(v, w)$. Jak daleko napravo od vrcholu v se růžový vrchol w zastaví je dáno délkou hrany (v, w). Jelikož jsme předpokládali, že tato délka je *nezáporná*, nemůže se nikdy stát, že by se pohybující se růžový vrchol w zastavil až vlevo od červeného vrcholu v. Toto je místo, kdy je předpoklad nezápornosti klíčový pro správnou funkci algoritmu.

Pokud byl již růžový vrchol w dosažený, pak závisí na tom, zda se změní hodnota $E(w)$. Pokud se nezmění, růžový vrchol w se nepohne; v opačném případě se posune doleva, což graficky vyjadřuje snížení hodnoty $E(w)$ na $E(v) + \ell(v, w)$. Z toho je ihned vidět, že se sice růžový vrchol posune doleva, ale ani v tomto případě se nemůže dostat doleva od červeného vrcholu v.

Jakmile je červený vrchol probrán, posune se zleva hranice tmavě modré oblasti (která na začátku neexistovala). Tmavě modrá oblast zahrnuje probrané vrcholy.

Nyní je velmi názorně vidět, že ohodnocení E vrcholů se mění velmi přirozeným způsobem:

1. pro nedosažené vrcholy v (ve světlém pásu vpravo) není $E(v)$ definováno;

2. pro dosažené vrcholy (střední pás) a probrané vrcholy (tmavý pás vlevo) je $E(v)$ definováno;

3. hodnota $E(u)$ libovolného probraného vrcholu u je menší nebo rovna hodnotě $E(v)$ libovolného dosaženého vrcholu v;

4. hodnoty $E(u)$ probraných vrcholů u v tmavě modré oblasti vlevo se nemění a jsou uspořádány v pořadí ve kterém se vrcholy stávaly probranými;

5. pro libovolný probraný vrchol u je $E(u)$ rovno $D(u)$, vzdálenosti u od počátku;

6. pro dosažený vrchol v s *minimálním* $E(v)$ mezi dosaženými vrcholy v je *také* $E(v) = D(v)$.

Provádějte si výpočet a sledujte, že uvedená tvrzení skutečně platí. Zůstaňte zde, dokud vám nebudou zcela přirozená a zřejmá.

Tvrzení 1 a 2 plynou ihned z popisu algoritmu: inicializace hodnoty $E(v)$ nastává současně se změnou klasifikace vrcholu.

Tvrzení 3 je triviálně platné po provedení inicializace a stačí ukázat, že se činnostmi spojenými s probráním vrcholu v neporuší a tedy zůstane v platnosti až do konce výpočtu. V okamžiku, kdy je zvolen a označen červenou barvou jistý dosažený vrchol v, je jeho odhad $E(v)$ minimální mezi dosaženými vrcholy. Během zpracování červeného vrcholu v se odhady E ostatních dosažených vrcholů mohou měnit a mohou přibývat nové dosažené vrcholy, ale, jak již bylo uvedeno výše, žádný z takových vrcholů se nemůže posunout vlevo od aktivního červeného vrcholu v a proto odhad tohoto vrcholu zůstává minimální až do konce zpracování, kdy je přesunut mezi probrané vrcholy.

Nově probraný vrchol proto má svůj odhad menší nebo roven než je odhad pro vrcholy, které zůstávají dosažené, ale neprobrané, takže se tvrzení 3 neporuší. Navíc odhad červeného probíraného vrcholu slouží jako "zarážka" pro hodnoty odhadů ostatních dosažených vrcholů, které nemohou klesnout na menší hodnotu (jinak řečeno tyto vrcholy se nemohou dostat vlevo od červeného vrcholu na obrazovce), ale přitom probrané vrcholy jsou všechny vlevo od zarážky (nebo maximálně na její úrovni) a mají proto svůj odhad nejvýše rovný odhadu pro zarážku.

Z podmínky 3 a nezápornosti cen hran ihned vyplývá, že podmínka

$$E(w) > E(v) + \ell(v, w)$$

při probírání dosaženého vrcholu v nebude *nikdy* splněna pro probraný vrchol w.

Z právě popsaného výkladu také vyplývá platnost tvrzení 4: když je vrchol v zařazován mezi probrané, byl ještě před chvilkou dosažený a jeho odhad byl tedy alespoň tak velký jako největší dosavadní odhad pro probraný vrchol.

Zdůvodnit poslední dvě tvrzení je složitější; uděláme to v následujících scénách a zatím jen sledujte, že jsou platné. Možná vás již při tom napadne proč.

Scéna: *Odhad vzdálenosti*

V této scéně budeme provádět stejný výpočet s využitím stejné grafické úpravy a animací jako ve scéně předchozí, ale budeme sledovat jinou vlastnost výpočtu. Z každého probraného nebo dosaženého vrcholu (s výjimkou počátku) vychází bílá tlustá zpětná šipka a z libovolného probraného nebo dosaženého vrcholu je podle těchto šipek možno jít až do počátku. Proti směru bílých šipek pak z počátku do daného vrcholu vede jednoznačně určená cesta, které budeme říkat *standardní* cesta. V této scéně si budeme všímat toho, že

- pro každý probraný nebo dosažený vrchol u je $E(u)$ rovno délce standardní cesty do u.

Současně se čtením následujícího textu si procházejte na obrazovce výpočet Dijkstrova algoritmu a sledujte, jak se tam vše projevuje.

Na začátku výpočtu tvrzení triviálně platí, protože se týká pouze počátku. Dokážeme, že se nikdy nemůže pokazit. Sledujme, co se děje při probírání vrcholu v, předpokládajíce, že když byl vrchol v vybrán, tvrzení platilo. Mimo jiné to znamená, že standardní cesta z počátku do vrcholu v má délku $E(v)$. Uvažujme nyní probírání nějaké hrany (v, w).

Jestliže byl vrchol w dosud nedosažený, bílá šipka z něj dosud nevedla a je algoritmem nastavena tak, že ukazuje na v, takže spolu s bílými šipkami z v do počátku dává standardní cestu do w, která je složením standardní cesty do v, která má délku $E(v)$, a probírané hrany, a má tedy délku $E(v)+\ell(v, w)$, což je přesně tak jak je nastavena počáteční hodnota $E(w)$.

Jestliže vrchol w byl již dosažený, pak do něho z počátku vede standardní cesta délky $E(w)$. Tuto cestu je třeba porovnat s cestou zmiňovanou v předchozím odstavci, která je složením standardní cesty z počátku do v a následující hrany (v, w) a má tedy délku $E(v) + \ell(v, w)$. Algoritmus porovná čísla vyjadřující jejich délky, a je-li druhá z nich kratší, přesměruje bílou šipku z vrcholu w tak, aby nám ukazovala nově nalezenou a výhodnější standardní cestu.

Zkuste si několikrát výpočet pro různé grafy s sledujte, jak se právě vysvětlená funkce algoritmu konkrétně projevuje.

Scéna: *Definitivní cesty*

Ještě jednou projdeme celý výpočet; scéna je podobná té předchozí, ale budeme se věnovat jinému důležitému pojmu. Cestu z počátku do nějakého

vrcholu budeme nazývat *definitivní*, jestliže je celá s případnou výjimkou posledního vrcholu u celá tvořena *probranými* vrcholy. Výjimka týkající se posledního vrcholu se zavádí proto, abychom mohli zkoumat i definitivní cesty do dosažených vrcholů.

Do nedosažených vrcholů definitivní cesty vést nemohou: předposlední vrchol cesty musí být probraný, ale z probraného vrcholu v nikdy nevede hrana do nedosaženého vrcholu. Než totiž byl vrchol v označen za probraný, konce hran z něho vycházejících musely být překlasifikovány na dosažené, pokud již dosažené nebo probrané nebyly předtím.

Cílem této scény je ukázat, že v klíčových okamžicích výpočtu je příkladem definitivní cesty do nějakého dosaženého nebo probraného vrcholu standardní cesta.

Jak jsem již naznačil, toto tvrzení nebude platit stále, ale bude splněno vždy v okamžiku, kdy červený aktivní vrchol byl překlasifikován na probraný a pohlcen zleva se rozšiřující tmavě modrou oblastí, ale algoritmus ještě neprovedl žádnou další akci. Takový okamžik budeme označovat za *kontrolní okamžik*; tento termín se objeví na obrazovce pod názvem scény, aby vám napověděl, kdy kontrolní okamžik nastává.

- V libovolném kontrolním okamžiku je standardní cesta do libovolného probraného nebo dosaženého vrcholu definitivní.

Důkaz je snadný: sledujte v jakých vrcholech končí bílé šipky. Všechny bílé šipky vždy končí buď v černých probraných vrcholech nebo v červeném vrcholu, který je právě probírán. Když je totiž bílá šipka vytvořena a nebo přesměrována, vede do červeného aktivního vrcholu, který posléze je překlasifikován na černý probraný a jeho klasifikace se pak již nemění.

V kontrolním okamžiku červený vrchol změní barvu na černou je prohlášen za probraný. V tomto okamžiku tedy *všechny* bílé šipky končí v probraných vrcholech. Jelikož s výjimkou posledního vrcholu standardní cesty na všechny její ostatní vrcholy ukazuje bílá šipka, je tedy v kontrolním okamžiku standardní cesta definitivní.

Projděte si jednou nebo dvakrát výpočet a sledujte, jak se právě popsané jevy konkrétně projevují.

Scéna: *Správnost Dijkstrova algoritmu*

Nyní se podíváme na stejnou scénu naposledy a ověříme, že platí

- V libovolném kontrolním okamžiku je pro každý probraný nebo dosažený vrchol v délka nejkratší definitivní cesty z počátku do v rovna $E(v)$.

V minulých dvou scénách jsme si ukazovali, že $E(v)$ je rovno délce standardní cesty do v, která je v kontrolních okamžicích definitivní. Je proto jenom třeba ověřit, že do v nevede definitivní cesta, která je kratší.

Na začátku výpočtu je to z triviálních důvodů pravda: jediná definitivní cesta je cesta obsahující samotný počátek a žádnou hranu, má délku 0 a přitom $E(\text{počátek}) = 0$.

Je tedy třeba ukázat, že pokud by se platnost tvrzení v průběhu výpočtu na chvilku porušila, do nejbližšího kontrolního okamžiku se vše zase spraví.

Co by se mohlo přihodit, aby tvrzení přestalo platit: v okamžiku, kdy aktivní červený vrchol se stane probraným a zčerná, může vzniknout jedna nebo více definitivních cest a některá z nich by mohla být kratší než je $E(w)$, kde w je její koncový vrchol.

Krokujte si znovu algoritmus; v této scéně se v různých okamžicích objevují barevně zvýrazněné cesty do různých vrcholů, které mají všechny následující vlastnost: v daném okamžiku ještě nejsou definitivní, ale stanou se definitivními, jakmile aktivní červený vrchol se stane probraným a zčerná. Pokud je taková cesta znázorněna, budete na to upozorněni v podtitulku. Takových cest obvykle je v daném okamžiku více; pokud je tomu tak, pak knoflíkem **Jiná cesta** je možné Algovizi požádat o zvýraznění jiné možné cesty (výběr je prováděn náhodně).

Každá taková cesta musí obsahovat červený aktivní vrchol, který zatím není, ale brzo bude probraný. Červený vrchol přitom není na konci cesty, protože definitivnost cesty nezávisí na stavu posledního vrcholu.

Situace na obrazovce, které při výpočtu nastávají, pokrývají dvě hlavní možnosti, které mohou nastat a uvidíme, že ani v jednom případě nevznikne definitivní cesta, porušující naše tvrzení. V podtitulku se pro vaši orientaci objeví indikace, které ze dvou možností nastává. Hrany úseku cesty od počátku do červeného aktivního vrcholu (který je v tomto okamžiku ještě stále dosažený, ale neprobraný) se zbarví fialově, barva ostatních hran bude závislá na situaci popsané dále.

Možnost 1: Za červeným aktivním vrcholem v cestě leží probraný vrchol. Uvažovaná cesta se zvýrazní následujícím trochu složitým způsobem: Probraný vrchol, ležící bezprostředně za červeným aktivním vrcholem se obarví fialově; může, ale nemusí být posledním vrcholem cesty. Hrany cesty od počátku po červený aktivní vrchol jsou červené, hrana z červeného do fialového vrcholu se zbarví fialově a případné další hrany cesty zeleně. Červený vrchol označíme jako v a fialový za ním jako x. Poslední vrchol cesty označme jako z. Vrchol z může být různý od x, ale také může být vrcholu x roven; pak by zelený úsek na cestě chyběl.

Sledujte nyní červeno-fialovou cestu do fialového vrcholu x. Jelikož fialová hrana má nezápornou délku (zde znovu potřebujeme aplikační podmínku Dijkstrova algoritmu), má červeno-fialovrá cesta délku větší nebo rovnou délce červeného úseku do červeného aktivního vrcholu v (který je o fialovou hranu kratší). Červený úsek je ale definitivní cesta - nevadí, že obsahuje dosud neprobraný červený vrchol v, protože to je její konec. Proto je délka červeného úseku

větší nebo rovna $E(v)$ podle indukčního předpokladu, ale platí $E(v) \geq E(x)$, protože v je zatím jen dosažený, ale x je probraný. Nakonec ale $E(x)$ je délka standardní cesty z počátku do x; je to cesta určená bílými šipkami.

Červeno-fialová cesta do x, která definitivní teprve bude, tedy není kratší než standardní cesta do x délky $E(x)$, která již definitivní je. Pokud námi uvažovaná cesta nekončí ve x ale v nějakém vrcholu z, pak také červeno-fialovo-zelená cesta (zatím nedefinitivní), která je složením červeného úseku délky alespoň $E(x)$, fialové hrany a zeleného úseku, není kratší než složení standardní cesty z počátku do x proti směru bílých šipek s délkou přesně $E(x)$ a zeleného úseku. Posledně jmenovaná cesta ale již je definitivní cestou do z a tedy není kratší než $E(z)$ a proto kratší než $E(z)$ není ani červeno-fialovo-(zelená) cesta do z.

Budoucí definitivní cesta, ve které za aktivním červeným vrcholem ještě leží alespoň jeden probraný vrchol, tedy nemůže zkoumané tvrzení ohrozit.

Možnost 2: Červený aktivní vrchol je předposlední vrchol cesty.
Je zvýrazněna cesta procházející přes černé probrané vrcholy a přes červený aktivní vrchol v do nějakého vrcholu w. Taková cesta může mít skutečně délku menší než je $E(w)$.

Uvažujeme okamžik, kdy algoritmus v rámci zpracovávání červeného aktivního vrcholu v bude probírat hranu (v, w), která zčervená. Délka naší cesty je nejméně $E(v) + \ell(v, w)$, protože její úsek od počátku po vrchol v je definitivní cesta do v a tedy nemůže být kratší než $E(v)$. Pokud by délka cesty byla *menší* než $E(w)$, pak by platilo $E(w) > E(v) + \ell(v, w)$, což je přesně test, který algoritmus provede, když v rámci zpracovávání červeného aktivního vrcholu v začne probírat hranu (v, w). Když je test splněn, pak $E(w)$ sníží na hodnotu $E(v)+\ell(v, w)$, tedy tak, že uvažovaná cesta (původně kratší než $E(w)$) již nebude kratší než $E(w)$.

Algoritmus je tedy navržen přesně tak, že si povšimne, že hrozí vznik definitivní cesty kratší než $E(w)$ a předejde tomu patřičnou modifikací hodnoty $E(w)$. Když pak dojde ke kontrolnímu okamžiku, vše je již zase v naprostém pořádku.

Povšimněte si, že právě probraná možnost zahrnuje také případ, kdy poslední vrchol cesty je v uvažovaném okamžiku *nedosažený*, protože červený vrchol dosud není probraný. Než se ale probraným stane, bude poslední hrana cesty probrána a přitom bude konec cesty překlasifikován na dosažený a odhad E pro vrchol bude nastaven přesně tak, aby tvrzení nebylo porušeno.

Jistě vám je jasné, proč jsme tvrzení dokazovali: na konci výpočtu jsou *všechny* vrcholy probrané a tedy *všechny* cesty jsou definitivní. Pak ale tvrzení prostě říká, že po ukončení výpočtu je pro každý vrchol v číslo $E(v)$ rovno délce nejkratší cesty z počátku do v, což je to, co jsme o Dijkstrově algoritmu chtěli dokázat.

14.3 Bellman-Fordův algoritmus

Dijkstrův algoritmus a Bellman-Fordův algoritmus jsou si formálně velmi podobné. Oba vycházejí z prohledávání grafu, při kterém si dosažené (ale neprobrané) vrcholy řadíme do fronty. Odlišnost spočívá v manipulaci s daty ve frontě. U Bellman-Fordova algoritmu se používá klasická FIFO fronta, zatímco Dijkstrův algoritmus využívá prioritní frontu, ze které se vybírá vrchol, minimalizující jistou hodnotu.

Tato z formálního hlediska nepodstatná odlišnost vede k velkému rozdílu chování. Hlavní rozdíl je ten, že při výpočtu podle Dijkstrova algoritmu je dosažený vrchol probírán v okamžiku, kdy $E(u)$ je délka nejkratší cesty, takže již nikdy nebude muset být měněna, zatímco u Bellman-Fordova algoritmu se hodnota $E(u)$ probraného vrcholu často mění (sníží) a tedy všechny údaje, které z původní hodnoty $E(u)$ byly odvozeny, pozbývají platnosti a musejí být přepočítány, takže výpočet Bellman-Fordova algoritmu někdy trvá poměrně dlouho.

Není to ovšem vada algoritmu plynoucí z použití primitivnější fronty, ale cena za to, že si troufne na obecnější grafy se záporně ohodnocenými hranami (lepší fronta by v takovém případě stejně příliš nepomohla).

Scéna: *Dosažitelný záporný cyklus*

Cílem této scény je ukázat, proč dosažitelný záporný cyklus vadí při definici nejkratší cesty. Je to závada principiální: nebylo by možné rozumně definovat, co vlastně je nejkratší cesta, takže bychom nebyli schopni říci, co vlastně má algoritmus počítat.

Zobrazený graf obsahuje *záporný cyklus*, to znamená orientovaný cyklus, pro který součet cen jeho hran je záporný. Cyklus je možno zobrazit tak, že se zaškrtne checkbox **Záporný cyklus** - jeho hrany zčervenají. Cyklus je navíc dosažitelný orientovanou cestou z počátku, například zeleně označená cesta, která nemusí být jediná.

Knoflíkem **Krok** je možno cestovat z počátku po zelené cestě k zápornému cyklu a pak jej obíhat. V pravém horním rohu se neustále zobrazuje cena cesty, kterou cestovatel urazil. (Změna volby z **Krátký krok** na **Dlouhý krok** vede k cestování po skocích - prvním do vstupního bodu cyklu a každým dalším jeden celý oběh cyklu).

Za každý oběh cestovatel "zaplatí" zápornou částku, přeloženo do normální řeči získá jistý stále stejný obnos. "Cena" zaplacená za proběhnutou dráhu tedy každým oběhem klesá a po dostatečné době poklesne pod libovolnou předem stanovenou hranici.

Do vrcholů dosažitelného záporného cyklu tedy neexistuje nejlevnější cesta - každou cestu lze učinit ještě levnější dalším oběhem.

Existují různé varianty Bellman-Fordova algoritmu, které se v přítomnosti záporného cyklu chovají různě. Nejjednodušší a naivní varianta spoléhá na to,

že ve vstupním grafu dosažitelný záporný cykl není, a pokud je podvedena, pak se nikdy nezastaví v marné snaze nalézt nejkratší cesty do vrcholů.

Trochu lepší varianta si spočítá, jak dlouho by měl výpočet trvat, kdyby v grafu dané velikosti nebyl dosažitelný záporný cyklus. Pokud je tento limit překročen a výpočet trvá déle, násilně se zastaví. V takovém případě sice víme, že v grafu dosažitelný záporný cyklus je, ale nevíme kde.

Nakonec nejdokonalejší varianty případný dosažitelný záporný cyklus nejenom detekují, ale i přesně popíší.

Bellman-Fordův algoritmus se velmi často používá nikoli k hledání nejkratších cest, ale pro detekci a identifikaci případného záporného cyklu. Příkladem takové aplikace je hledání maximálního toku minimální ceny v síti, které ale v této knize popisováno není.

Graf v této scéně je opět možno měnit, ale s podmínkou, že změna zachovává přítomnost alespoň jednoho dosažitelného záporného cyklu.

Je zřejmé, že algoritmus pracuje pouze v množině vrcholů dosažitelných z počátku. Přítomnost záporného cyklu nedosažitelného z počátku tedy nevadí, v takovém případě je možno nejkratší cesty korektně definovat a algoritmus je správně určí.

Scéna: *Graf bez záporného cyklu*

Nyní si probereme výpočet Bellman-Fordova algoritmu v grafu, ve kterém je každý vrchol dosažitelný z počátku, ale který neobsahuje záporný cyklus.

Algoritmus bude na první pohled připomínat prohledávání do hloubky a také Dijkstrův algoritmu a teprve později zjistíme, jak hluboce se od nich liší. Podobně jako oba uvedené algoritmy bude rozdělovat vrcholy na nedosažené, dosažené (ale ještě neprobrané) a probrané.

Podobně jako u Dijkstrova algoritmu budeme pro každý dosažený nebo probraný vrchol v mít určen odhad $E(v)$, který na konci výpočtu bude označovat délku nejkratší cesty z počátku do v. Liší se ale výběr vrcholu určeného k probírání - budeme stejně jako u prohledávání do hloubky používat FIFO frontu, ve které budou dosažené vrcholy, kterou ale budeme používat trochu komplikovanějším způsobem. Navíc budeme dosažené vrcholy, které se nacházejí ve frontě dělit na nové a staré. Toto dělení není nutné pro samotný výpočet, ale bude se nám hodit při úvahách o výpočetní rychlosti tohoto způsobu (pokud tedy algoritmus chcete programovat, dělení na staré a nové a s tím související operace můžete vyhodit - jsou popsány menším písmem).

počátek v_0 se označí jako dosažený, vloží se do původně prázdné fronty jako nový vrchol a položí se $E(v_0) \leftarrow 0$;
ostatní vrcholy se označí jako nedosažené;
pak dokud existuje alespoň jeden dosažený (ale neprobraný) vrchol, opakuje se následující činnost:

> pokud jsou všechny vrcholy ve frontě označeny jako nové
> překlasifikují se na staré;
> vrchol, který stojí ve frontě na prvním místě se označí se jako v;
> pro každou hranu (v, w) vycházející z v se provede následující:
> jestliže w je nedosažený, pak se překlasifikuje na nový dosažený
> a položí se $E(w) \leftarrow E(v) + \ell(v, w)$;
> jinak pokud $E(w) > E(v) + \ell(v, w)$, tak se pro w provede:
> položí se $E(w) \leftarrow E(v) + \ell(v, w)$;
> jestliže w byl probraný, překlasifikuje se na nový dosažený
> [a dá na konec fronty];
> v se vyjme z fronty a označí se za probraný.

V každém kroku algoritmus vybere nejdéle známý dosažený vrchol a označí jej jako *aktivní*. Barevné označení vrcholů bude stejné jako u Dijkstrova algoritmu: nedosažený vrchol žlutý, dosažený zelený, probraný vrchol černý, neprobraná hrana modrá a probraná černá. Výjimky z tohoto pravidla budou počátek (na začátku modrý), aktivní vrchol v bude červený a konec w probírané hrany vycházející z červeného v bude růžový. Kromě toho zelené dosažené vrcholy budou ještě barvou děleny na tmavě zelené staré a světle zelené nové. U každé hrany je poznamenána její cena, u probraných a dosažených vrcholů je uvedena hodnota $E(v)$. Používáme i tlusté bílé šipky, ukazující, odkud jsme se do vrcholu dostali dosud nejvýhodnějším způsobem.

Uvedený algoritmus má dvě varianty podle toho, zda uvažujeme text v hranatých závorkách. Jestliže je vrchol w dosažený, ale v průběhu zpracování hrany (v, w) dojde ke snížení $E(w)$, pak v jedné variantě (ignorující text v hranatých závorkách) se ponechá vrchol w na jeho místě ve frontě s novou hodnotou $E(w)$, zatímco v druhé variantě (uvažující text v hranatých závorkách) se vrchol w vyjme z fronty a s novou hodnotou $E(w)$ se zařadí na její konec, takže k jeho zpracování dojde později.

Není zřejmé, která varianta je výhodnější. Přeřazení na konec je vhodné, pokud ani nová hodnota odhadu $E(w)$ ještě není konečná, protože v takovém případě zpracování vrcholu i případné následné akce jsou zbytečná práce, která bude opakována znovu později s obecně lepší hodnotou odhadu, takže odložení probírání vrcholu je na místě. Pokud je ale odhad již roven konečné hodnotě, je lepší ponechat vrchol na jeho místě vpředu ve frontě a tedy jej zpracovat co nejdříve. Která možnost ale nastává je v průběhu výpočtu těžké poznat.

Zkuste si výpočet algoritmu projít; je možno nechat zobrazit zjednodušený

program, ve kterém je zvýrazněn právě prováděný příkaz.

Scéna: *Fáze*

Scéna je takřka stejná jako předchozí, ale slouží pro rozdělení výpočtu na *fáze*, což bude výhodné pro zkoumání výpočetní doby.

Ukážeme totiž, že pokud v grafu není dosažitelný záporný cyklus, pak se provede nejvýše n fází, kde n je počet vrcholů grafu a v každé fázi bude každý vrchol a každá hrana probírány nejvýše jednou. Z toho ihned vyplyne, že výpočet bude proveden v čase $O(n(n+m))$, kde m je počet hran grafu.

Rozdělení na fáze je důvodem, proč jsme dosažené vrcholy v algoritmu dělili na staré a nové. Je vidět, že ve frontě staré vrcholy předchází nové vrcholy a kdykoli je do fronty přidáván vrchol, je přidáván jako nový. Jestliže se těsně před výběrem prvního vrcholu fronty za aktivní zjistí, že ve frontě jsou jen nové vrcholy, pak budou překlasifikovány na staré (a ztmavnou) a tím začíná *nová fáze* výpočtu. Při krokování výpočtu bude přebarvení vrcholů a začátek nové fáze a pořadové číslo právě prováděné fáze explicitně oznámeno.

Je jasné, že v rámci jedné fáze vrchol může být vybrán jako aktivní a být probírán nejvýše jednou a v rámci jeho probírání se každá z něho vycházející hrana probírá právě jednou. Kdyby byl vrchol z fronty odebrán a v rámci téže fáze později do fronty znovu zařazen, bude označen za nový a bude proto aktivován nejdříve v následující fázi. Doba trvání jedné fáze je tedy $O(n+m)$.

Projděte si výpočet a sledujte začátky a konce fází. Nepřecházejte do další scény, pokud nebudete schopni určit konce fází bez sledování hlášení.

Scéna: *Průběh fází*

V této scéně budeme uvažovat variantu, která v případě zlepšení odhadu pro dosažený vrchol tento vrchol "vykopne" na konec fronty jako nový dosažený vrchol. U každého dosaženého nebo probraného vrcholu v budeme uvádět nejen hodnotu odhadu ceny nejlepší cesty do vrcholu, ale i cenu $D(v)$ optimální cesty, tedy dvojici $E(v) : D(v)$.

Na začátku scény nebo po každé změně grafu se zobrazí stav grafu po inicializaci výpočtu. Scéna se prochází následujícím způsobem. Stiskněte knoflík **Další.** Průběh výpočtu jsme již viděli v předchozí scéně a tak se znázorní přímo výsledek výpočtu. Objeví se mimo jiné bílé šipky na předchůdce, ukazující optimální cesty do vrcholů stejně tak jak tomu bylo v kapitolách o prohledávání a o Dijkstrově algoritmu.

Po dalším stisknutí knoflíku **Další** se barevné označení a ohodnocení vrcholů a hran grafu se nezmění, ale graf se překreslí tak, že se vrcholy horizontálně přemístí tak, aby vytvořily sloupce, kterým budeme říkat *vrstvy* a které jsou zvýrazněny pruhy na pozadí. Vrstvy jsou určeny tak, že počátek je ve vrstvě zcela vlevo a každá bílá šipka vede z jisté vrstvy do vrstvy, která s ní sousedí vlevo.

Jinými slovy řečeno je každý vrchol v tolikáté vrstvě zleva, kolik hran je na standardní cestě do tohoto vrcholu. Počátek je tedy ve vrstvě 0 a jde-li z vrcholu w bílá šipka do vrcholu v, který je v k-té vrstvě, pak w je ve vrstvě $k+1$.

Nyní zmáčkněte znovu knoflík **Další.** Barvy vrcholů a hran a čísla spojená s vrcholy se vrátí do stavu před začátkem výpočtu, ale polohy vrcholů, dané rozdělením do vrstev, se již nezmění.

Nyní začněte výpočet krokovat knoflíkem **Krok** ještě jednou. Uvidíte, že po rozdělení do vrstev se akce, které algoritmus provádí, stanou snadno pochopitelné a přehledné. Navíc zjistíte, že "kdybychom to bývali byli věděli", mohli bychom si ušetřit mnoho práce, která se v překresleném grafu stane na první pohled zbytečná. Bohužel se jedná o znalost "generála po bitvě"; informace o tom, které práce nebylo nutné provádět, se stane zřejmou až poté, kdy je výpočet dokončen a je možno snadno určit rozdělení do vrstev.

Během výpočtu budou pruhy na pozadí, nesoucí vrstvy vrcholů, označeny barvami, které vystihují roli vrcholů vrstvy ve výpočtu. Povšimněte si především, že ke změně barev pruhů dojde vždy při přechodu mezi dvěma fázemi. Změna barev bude spočívat v přesunu barev o jednu vrstvu doprava.

Nebude na škodu, když si nyní projdete výpočet a budete sledovat jak jsou posuny barev synchronizovány se změnami fází, aniž byste věděli co jednotlivé vrstvy znamenají a jejich barvy znamenají.

Pozorování, které jste právě ověřovali, ukazuje, že počet fází není větší než počet vrstev. Jelikož každá vrstva obsahuje alespoň jeden vrchol, je vrstev nejvýše tolik, kolik je vrcholů. Spolu s poznatkem o době potřebné k provedení fáze to dává celkový časový odhad $O(n(n+m))$ pro dobu výpočtu v nepřítomnosti dosažitelného záporného cyklu. V mnoha případech je strom bílých šipek dosti široký, neboli mnohé vrstvy obsahují hodně vrcholů a proto může počet vrstev být výrazně menší než počet vrcholů.

Jelikož dosažitelný záporný cyklus způsobí, že se výpočet algoritmu v jeho základní podobě nikdy nezastaví, je možno algoritmus upravit takto: pokud by se začala provádět $(n+1)$-ní fáze, je možno výpočet ukončit, graf obsahuje dosažitelný záporný cyklus. Později si ukážeme, jak po zastavení takový cyklus také snadno najít, je-li to požadováno. Existují ale i jiné varianty algoritmu, které umí detekovat záporný cyklus podstatně rychleji na základě jiných jevů, které jej doprovázejí.

Vraťte nyní výpočet znovu na začátek knoflíkem **Reset** a provádějte jej opětovně krok po kroku a sledujte, že probíhá jak je popsáno dále.

Kromě zahájení a ukončení výpočtu je vždy jedna vrstva označena fialovou barvou; tato vrstva se nazývá *aktivní.* Název je odvozen od toho, že v této vrstvě se bude často nacházet vybraný červený aktivní vrchol. Pokud aktivní vrstva není zcela napravo, pak bezprostředně vpravo od ní je vrstva obarvená zelenou barvou a nazývá se *přijímající* vrstva. Vrstvy nalevo od aktivní vrstvy

(existují-li) jsou tmavě modré a jsou označovány jako *probrané* vrstvy, zatímco vrstvy napravo od přijímající vrstvy (existují-li) jsou světle modré *vzdálené* vrstvy.

Vrchol v nazveme *definitivní*, právě když platí $E(v) = D(v)$, neboli odhad $E(v)$ ceny cesty do vrcholu v má již svou konečnou hodnotu rovnou ceně nejlacinější cesty do v. Jestliže definitivní vrchol v je probraný, pak již nemůže znovu být označen za dosažený, protože změna klasifikace z probraného na dosažený se děje současně s poklesem jeho odhadu $E(v)$, což ale již nemůže nastat. Dosažený definitivní vrchol ze stejných důvodů nemůže být "vykopnut" na konec fronty ani ve variantě algoritmu, která to připouští. Bílá šipka v takovém případě již ukazuje předchůdce na optimální cestě.

Nyní provádějte znovu a velmi pomalu krokování výpočtu a sledujte, že probíhá jak je popsáno dále:

- všechny vrcholy v probraných vrstvách jsou probrané a definitivní;
- vrcholy v aktivní vrstvě jsou na začátku fáze *všechny* dosažené a definitivní; během prováděné fáze budou probrány a jednou provždy přeřazeny mezi probrané vrcholy;
- každý vrchol v přijímající vrstvě nebo ve vzdálených vrstvách se může nacházet v libovolném ze tří stavů, ale *není* definitivní;
- každý vrchol přijímající vrstvy se po skončení fáze stane definitivním, zatímco každý vrchol vzdálené vrstvy bude i po ukončení fáze nedefinitivní.

Je tedy například vidět, že v přijímající vrstvě nebo ve vzdálených vrstvách mohou být i probrané vrcholy, ale jsou to ty, které určitě budou ještě alespoň jednou překlasifikovány na dosažené a později znovu probírány. Jejich zpracování jsme si tedy mohli odpustit, kdybychom to bývali byli věděli.

Na druhé straně vrcholy v aktivní vrstvě jsou ty, jejichž probírání je užitečné, protože jsou definitivní a jejich odhad se již určitě nikdy nezmění. Po probrání celé aktivní vrstvy budou všechny její vrcholy po právu zařazeny mezi zcela vyhaslé probrané vrcholy v probraných vrstvách.

Nyní již asi rozumíte nejen tomu, co se při výpočtu děje, ale i *proč* k tomu dochází. I přesto může být užitečné si projít výpočet ještě jednou a sledovat výpočet s plným porozuměním:

Jak bylo řečeno výše, definitivní probraný vrchol se již nikdy nestane dosaženým a proto v probraných vrstvách nedochází k žádné aktivitě. Navíc definitivní dosažený vrchol nikdy nepozbude svého místa ve frontě, takže všechny vrcholy aktivní vrstvy budou v dané fázi probrány a stanou se definitivními probranými.

Nechť w je nyní vrchol přijímající vrstvy. Podívejme se nejprve na situaci v grafu, jak vypadala na konci výpočtu. Uvažujme standardní cestu do w, jak vypadala v ten okamžik vypadala. Z w vede bílá šipka do předposledního vrcholu této standardní cesty, a tento vrchol označíme v. Podle definice vrstev tedy leží v o jednu vrstvu blíže k počátku. Pokud $D(w)$ je nejkratší vzdálenost z počátku do w a $D(v)$ je nejkratší vzdálenost z počátku do v, pak určitě $D(w) = D(v) + \ell(v, w)$, protože hrana (v, w) je součástí standardní cesty.

Teď se vraťme do okamžiku, kdy w je v zelené přijímající vrstvě. Jelikož v je o 1 vrstvu vlevo, je ve fialové aktivní vrstvě a, jak jsme viděli výše, $E(v)$ již je proto rovno vzdálenosti $D(v)$ z počátku do v. Spolu s rovností z předchozího paragrafu tedy plyne, že v uvažovaný okamžik je $D(w) = E(v) + \ell(v, w)$. Jelikož je ale vrchol w dosud v zelené přijímající vrstvě, je také $E(w) > D(w)$.

Nejpozději při probírání vrcholu v, tedy stále v dané fázi, se tedy hodnota $E(w)$ změní na $D(w) = E(v) + \ell(v, w)$ a vrchol w se proto stane definitivním. V důsledku změny zároveň bude w zařazen do fronty jako nový dosažený ale již definitivní vrchol. Své místo ve frontě proto již si udrží a při přechodu do následující fáze se změní na starý dosažený definitivní vrchol aktivní vrstvy.

Nakonec si uvědomíme, že vrchol se může stát definitivním jen při probírání hrany z vrcholu, který je dosažený a již definitivní je. Je-li w vrchol vzdálené vrstvy, pak kdyby v rámci uvažované fáze stal definitivním, pak by se tak stalo při probírání některého vrcholu v aktivní vrstvy (která jediná obsahuje dosažené definitivní vrcholy). Ukazatel na předchůdce na optimální cestě (bílá šipka) by se pro vrchol w natrvalo nastavil tak, aby ukazoval na vrchol v, což by ale znamenalo, že w je ve vrstvě sousedící s aktivní vrstvou a bylo by ve sporu s předpokladem, že w je ve vzdálené vrstvě. Vrcholy vzdálených vrstev se tedy v rámci uvažované fáze nemohou stát definitivními.

Celkově tedy vidíme, že po ukončení fáze a posunutí aktivní a přijímající vrstvy doprava se opět dostaneme do výše popsané situace, kdy vrcholy aktivní vrstvy a vrstev nalevo od ní jsou definitivní, v ostatních vrstvách nedefinitivní, a vrcholy aktivní vrstvy jsou všechny připraveny ve frontě.

Výpočet tedy po překreslení grafu do vrstev probíhá tak, že aktivní vrstva, které se pohybuje zleva doprava mění vrcholy na definitivní, zatímco napravo od ní může docházet k provádění úkonů, které vycházejí z předběžných a neoptimálních údajů a proto nejsou schopny vypočítat byť jedinou užitečnou hodnotu.

Scéna: *Průběh fází ve velkém grafu*

Scéna ukazuje průběh výpočtu v pevném velkém grafu, navrženém tak, aby byl i po překreslení do vrstev dostatečně přehledný.

Výpočet se odstartuje knoflíkem **Výpočet** a je možné zastavit knoflíkem **Stůj,** znovu spustit knoflíkem **Výpočet** nebo jej vrátit zpět knoflíkem **Zpět.** V závislosti na volbě na ovladači je přitom graf nakreslen buď s původními po-

lohami vrcholů nebo překreslen do vrstev jako v předchozí scéně. Zkuste si obě varianty; zatímco při původním nakreslení se průběh výpočtu jeví chaotický, výpočet ve vrstveném nákresu velmi pěkně ukazuje jak aktivní vrstva mění vrcholy na definitivní, zatímco nepodstatná aktivita ve vzdálených vrstvách má stále chaotický charakter.

Scéna: *Průběh fází bez vykopnutí na konec*

Bellman-Fordův algoritmus ve své variantě s "vykopnutím" definitivního vrcholu na konec fronty, dojde-li ke zlepšení jeho odhadu měl výhodu v tom, že průběh výpočtu byl po překreslení do vrstev mimořádně přehledný. V této scéně se budeme zabývat průběhem výpočtu při použití algoritmu ve variantě bez vykopnutí, tedy dosažený vrchol za žádných okolností nemění své místo ve frontě dokud nedospěje na začátek a po probrání je odebrán.

Jak uvidíme, průběh výpočtu je podobný, ale některé vrcholy mohou ve svém vývoji "předbíhat fáze". Změna, která u této varianty může nastat, je následující:

Je-li na začátku fáze ve frontě dosažený vrchol w přijímající vrstvy, pak u něho dojde zaručeně k alespoň jedné z těchto dvou událostí:

- po probrání některého vrcholu z aktivní vrstvy se w stane definitivním;
- vrchol w je probírán.

Navíc se může stát, že k první z výše uvedených událostí dojde dříve. U varianty s vykopnutím by to znamenalo, že se w přesune jako nový dosažený na konec fronty a nebude probírán dříve, než v následující fázi. Ve variantě bez vykopnutí naopak zůstane na svém místě ve frontě jako starý dosažený a bude probrán ještě v uvažované fázi. Může dokonce dojít k tomu, že při probírání vrcholu w (který je v té době již definitivní) se v uvažované fázi vytvoří definitivní vrchol v první vzdálené vrstvě a pokud tento vrchol je dosažený, ale je ve frontě až za vrcholem w, bude ještě v rámci této fáze jako definitivní probírán a může vytvořit jiný definitivní vrchol v následující vrstvě atd.

Příklad uvedený v této scéně ukazuje, že v rámci fáze mohou vzniknout dosažené definitivní vrcholy i probrané definitivní vrcholy (tedy vrcholy v konečném vývojovém stádiu) nejen v přijímající vrstvě, ale i v libovolné vzdálené vrstvě.

Při zaškrtnutém checkboxu **Předbíhající** jsou vrcholy, které se staly definitivními v dřívější fázi, než by odpovídalo jejich zařazení do vrstev, označeny po přechodu do definitivního stavu barevným pozadím.

Je vidět, že odhad počtu fází pro variantu s vykopnutím je i odhadem pro variantu bez vykopnutí; druhá varianta může dokonce skončit s menším počtem fází. To ovšem neznamená, že by výpočet varianty bez vykopnutí

musel být rychlejší. Významnou část doby výpočtu představují neužitečné akce při probírání nedefinitivních vrcholů a navazující výpočty. Varianta bez vykopnutí může mít při stejném počtu fází větší průměrnou dobu pro ukončení fáze, neboť se nesnaží odkládat zpracování nedefinitivních vrcholů, což někdy provádí první varianta tím, že je přesouvá na konec. Z hlediska asymptotického chování v nejhorším případě jsou ale oba algoritmy rovnocenné.

Scéna: *Záporný cyklus*

Tato scéna ukáže jednoduchý důvod, proč se algoritmus ve výše uvedeném provedení (libovolná varianta) nemůže zastavit v případě, že je v grafu dosažitelný záporný cyklus. Graf je navržen tak, že takový cyklus obsahuje. Hrany cyklu jsou zvýrazněny použitím červeného obrysu.

Řekneme, že hrana (u, v) je *horká*, jestliže jsou oba její konce dosažené nebo probrané (a je pro ně tedy definováno E) a platí $E(u) + \ell(u, v) < E(v)$. Nyní si uvedeme tři jednoduchá pozorování:

- Z probraného vrcholu nevychází žádná horká hrana.
- Horká hrana může vycházet jen z dosaženého vrcholu.
- Jestliže všechny vrcholy záporného cyklu jsou dosažené nebo probrané, pak cyklus obsahuje alespoň jednu horkou hranu.

K důkazu prvního tvrzení si stačí uvědomit, že v okamžiku probírání vrcholu u u každé horké hrany (u, v) změníme $E(v)$ na hodnotu $E(u) + \ell(u, v)$ a poté se $E(v)$ může měnit jen tak, že se snižuje. Kromě toho pokud by došlo ke změně $E(u)$, přestane být vrchol u probraný.

Druhé tvrzení je okamžitý důsledek prvního. Výchozí vrchol horké hrany nemůže podle definice být nedosažený a podle prvního tvrzení probraný.

V třetím tvrzení potřebujeme, aby vrcholy nebyly nedosažené k tomu, aby odhady E pro jeho vrcholy byly definovány. Pro jeho důkaz si stačí napsat nerovnosti, které by pro cyklus tvořený vrcholy $w_1, w_2, \ldots, w_k$ platily, kdyby žádná jeho hrana nebyla horká:

$$E(w_1) + \ell(w_1, w_2) \geq E(w_2)$$

$$E(w_2) + \ell(w_2, w_3) \geq E(w_3)$$

$$\ldots$$

$$E(w_{k-1}) + \ell(w_1, w_k) \geq E(w_k)$$

$$E(w_k) + \ell(w_k, w_0) \geq E(w_1)$$

Sečtením těchto nerovností a odečtením výrazu $E(w_1) + \cdots + E(w_k)$ od obou stran bychom dostali nerovnost

$$\ell(w_1, w_2) + \ell(w_2, w_3) + \cdots + \ell(w_{k-1}, w_k) + \ell(w_k, w_1) \geq 0,$$

která by byla ve sporu se záporností cyklu.

Snadno se ověří, že po jisté době výpočtu algoritmu se všechny vrcholy dosažitelné z počátku stanou dosaženými nebo probranými. Pokud tedy v grafu je dosažitelný záporný cyklus, žádný jeho vrchol již nebude nedosažený, takže od tohoto okamžiku již bude v grafu stále alespoň jedna horká hrana. Pokud by se ale výpočet zastavil, tedy v grafu by nebyly dosažené vrcholy, pak by podle druhé vlastnosti žádná horká hrana v grafu nemohla zůstat.

Tato scéna během výpočtu ukazuje červenou barvou všechny horké hrany. Sledujte jak na začátku výpočtu postupně mizí nedosažené vrcholy a jaké je v dosažené oblasti rozložení horkých hran. V oblasti záporného cyklu je alespoň jedna horká hrana, která se vždy po probrání jejího výchozího vrcholu posune dále, ale nikdy nezmizí.

Ukázali jsme tedy, že výpočet Bellman-Fordova algoritmu se zastaví právě tehdy, když v grafu neexistuje záporný cyklus dosažitelný z počátku. Navíc pokud se výpočet zastaví, stane se to po nejvýše n fázích, kde n je počet vrcholů. To umožní poznat, zda se výpočet zastaví nebo poběží na věky. Způsobem nalezení dosažitelného záporného cyklu se v této knize nebudeme zabývat.

Kapitola 15

Minimální kostra grafu

Scéna: *Euklidovská kostra*

Scéna ukazuje množinu bodů v rovině, které budeme nazývat vrcholy. Naším cílem je propojit je nejlacinějším způsobem pomocí hran, které přímo spojují dvojice vrcholů. V této scéně je cena hrany dána Euklidovskou vzdáleností jejích vrcholů a cena propojení je rovna součtu cen použitých hran.

Poznamenávám, že Euklidovská metrika (označovaná také jako metrika $\mathcal{L}_2$) definuje vzdálenost d dvou vrcholů $s = [s_x, s_y]$ a $t = [t_x, t_y]$ takto:

$$d = \sqrt{(s_x - t_x)^2 + (s_y - t_y)^2}.$$

Po klepnutí na knoflík **Krok** Algovize ukáže nejlacinější propojení bodů tak, aby bylo možno se z každého vrcholu postupně dostat do libovolného jiného, takzvanou *minimální kostru*.

Změnou volby z **Počítej** na **Edituj** se přejde do editačního módu, kdy je možno stávající vrcholy přetahovat myší, vytvářet nové vrcholy a vynechávat již existující způsobem popsaným v závěru knihy. Po návratu do výpočetního režimu lze určit a zobrazit minimální kostru změněné množiny vrcholů.

Zkuste si měnit soubor vrcholů a sledovat, jak se mění jejich optimální propojení.

Výsledek propojení je zde i v následujících scénách vždy *strom*, to znamená souvislý graf bez cyklů. Souvislost požadujeme a acykličnost plyne z požadavku minimální ceny a z nezápornosti ceny propojení vrcholů: kdyby by ve výsledném propojení byl cykl, pak po vyhození jeho libovolné hrany by souvislost zůstala zachována (konce hrany by stále byly propojeny zbytkem cyklu), ale cena by ve sporu s minimalitou poklesla.

Jenom poznamenávám, že v některých scénách není cena propojení dvou vrcholů dána metrikou, ale zadaným číslem. Pak z právě uvedených důvodů

budeme požadovat, aby cena byla kladná.

Scéna: *Kostra v metrickém grafu*

Je opět dána množina vrcholů, ale některé dvojice jsou propojeny modrými spoji, kterým říkáme hrany. Opět hledáme minimální propojení vrcholů (bude nakresleno zeleně), ale nyní je povoleno propojovat jen dvojice spojené modrými hranami.

Je zřejmé, že pro existenci řešení je nutné, aby výchozí graf byl souvislý. Pokud graf není souvislý, Algovize protestuje a je nutno přepnout do editačního módu a buď přidat hrany nebo vynechat některé vrcholy. Kostra popsaná v předchozí scéně je speciální případ kostry grafu, kde zvolený graf je úplný.

Klikněte na **Krok** a Algovize ukáže řešení. Hrany grafu jsou úzké, hrany zvolené jako propojení jsou zdůrazněny podkladovým pásem zelené barvy. Knoflík **Zpět** řešení zase vymaže.

V editačním módu je graf možno měnit.

Scéna: *Kostra v ohodnoceném grafu*

Scéna je podobná předchozí, ale ceny hran grafu nejsou dány Euklidovskou vzdáleností koncových bodů, ale obecně kladnými čísly, připsanými u hran. Toto je nejobecnější případ; v praktických aplikacích znamená, že cena není dána prostou délkou spojení, ale i například charakterem terénu (např. telefonní kabel v rovné krajině, horách nebo na mořském dně) nebo způsobem provedení propojení.

Při přidávání nových hran je cena hrany zvolena náhodně mezi 1 až 99, ale dá se změnit. V appletu musí být cena hrany kladné celé číslo. Nulu a záporná čísla nepřipouštíme, protože by to změnilo charakter problému; omezení na celá čísla není na újmu obecnosti, ale z hlediska grafické úpravy je výhodné.

Hrajte si s množinou vrcholů, hranami a jejich cenami a sledujte, jak se tím mění minimální kostra grafu.

Scéna: *Obecný algoritmus*

Tato scéna ukazuje obecné schéma určování minimální kostry v případě bez omezení propojení jako v první scéně. Schéma připouští velkou volnost v lichých krocích výpočtu a konkrétní algoritmy pak tuto volnost omezují, vedeny obvykle snahou o jednoduchost nebo rychlost výpočtu. Libovolný výpočet spadající do schématu ale dává správný výsledek.

V této scéně předpokládáme cenu hran danou Euklidovskou délkou, případ obecných cen bude probrán dále. Množinu vrcholů lze upravovat stejným způsobem jako bylo uvedeno výše, změna ale resetuje výpočet.

Algoritmické schéma je velmi jednoduché a pracuje tak, že do původně prázdného grafu přidává postupně hrany tak, aby nevznikl cyklus a výpočet

se ukončí v okamžiku, kdy se vytvářený graf stane souvislým, neboli když se z něho stane strom.

V průběhu výpočtu je tedy neúplná kostra nesouvislý graf bez cyklů (les), který se rozpadá do řady komponent, které jsou stromy (souvislé grafy bez cyklů). Na začátku výpočtu jsou komponenty tvořeny jednotlivými vrcholy a přidáním každé hrany se dvě komponenty spojí v jednu a tedy se jejich počet sníží o 1. Původní množinu vrcholů spolu s množinou hran, které jsme již do vytvářené neúplné kostry přidali, budeme nazývat *výpočetní les*.

Pro odlišení komponent se také používá barevné označení vrcholů. Na začátku mají všechny vrcholy (představující různé jednovrcholové komponenty) různé barvy. Při spojení dvou komponent vrcholy jedné komponenty přijmou barvu vrcholů druhé z nich, takže vrcholy vzniklé komponenty budou zase mít stejnou barvu.

Jelikož tedy je na začátku N jednovrcholových komponent a na konci vše splyne v jednu, spočívá výpočet v $N-1$ opakováních následujících dvou kroků:

Krok: *Volba komponenty*

Algoritmus zvolí *libovolnou* komponentu $\mathcal{C}$ výpočetního lesa. $\diamond$

Krok: *Volba hrany*

Mezi dvojicemi (v, w), kde v je vrchol ve zvolené komponentě $\mathcal{C}$ a w vrchol mimo ni, vybereme dvojici s nejmenší vzdáleností a přidáme ji do výpočetního lesa. $\diamond$

Volba komponenty tedy poskytuje úplnou volnost, volba propojované dvojice naopak je zcela vynucená. Pouze v případě, že by existovalo několik dvojic stejné minimální vzdálenosti, bylo by povoleno zvolit libovolnou z nich.

Výběr komponenty se v appletu provádí v závislosti na volbě na ovladači. Je-li zvolena volba **Ručně,** pak volba komponenty se provede klepnutím na její libovolný vrchol. (Je ale také možno klepnout na knoflík **Zvol komponentu** a Algovize vybere komponentu náhodně.) Při volbě **Náhodně** Algovize provede volbu náhodně sama po stisknutí knoflíku **Krok.**

Krok určení minimální hrany provede Algovize vždy sama po trojím stisknutí knoflíku **Krok** (viz dále). Při volbě **Náhodně** tedy se výpočet pouze krokuje knoflíkem **Krok,** kdežto při ručním provádění volbu komponenty dělá uživatel sám (s tím, že ale může tuto povinnost předat Algovizi).

Ať je již komponenta zvolena jakkoli, její vrcholy se zvýrazní červenou barvou. Pro znázornění určování dvojice s minimální cenou se tato operace provádí ve třech krocích. V prvém z nich se provede animace, během níž se kolem vrcholů zvolené komponenty rozšiřují okolí, která mají všechna stejný poloměr, který se s časem zvětšuje. Jakmile se první bod některého z těchto okolí dotkne vrcholu ležícího mimo zvolenou komponentu, je nalezena nejkratší spojnice zvolené komponenty se zbývajícími vrcholy grafu. V druhém kroku (neanimovaném) okolí zmizí a zvolená hrana se zvýrazní červeně a v

třetím kroku se zvýraznění hrany odstraní (její červená barva se změní na standardní zelenou).

Scéna: *Metrika $\mathcal{L}_1$*

Scéna opět ukazuje výpočet kostry, ale vzdálenost dvou bodů není měřena Euklidovskou metrikou $\mathcal{L}_2$, nýbrž metrikou $\mathcal{L}_1$ (neboli součtovou metrikou), ve které je cena propojení dvou bodů se souřadnicemi $[x_1, y_1]$ a $[x_2, y_2]$ rovna

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

U této metriky nejsou okolí vrcholů kružnice, ale čtverce postavené na koso. Poznamenávám, že okolí bodu v je množina bodů vzdálených od v o nejvýše r, kde r je nějaké kladně reálné číslo; r se nazývá poloměr okolí.

Scéna: *Metrika $\mathcal{L}_{max}$*

Tato scéna je zase jako předchozí dvě, ale metrika, určující vzdálenost dvou bodů je tak zvaná maximová metrika $\mathcal{L}_{max}$ nebo $\mathcal{L}_\infty$, ve které je cena propojení dvou bodů se souřadnicemi $[x_1, y_1]$ a $[x_2, y_2]$ rovna

$$\max(|x_1 - x_2|, |y_1 - y_2|).$$

U této metriky nejsou okolí vrcholů kružnice, ale čtverce sedící na jedné své straně.

Scéna: *Metrický graf*

Tato scéna opakuje předchozí, cena hrany je opět její Euklidovská délka, ale propojující strom musí být zvolen z modrých hran výchozího grafu stejně jako v druhé scéně tohoto appletu.

Okolí vrcholů nejsou ukázána úplně, ale je jen znázorněno, kam až by dosahovala v nakreslených hranách. Nejbližší soused z jiné komponenty je nalezen, když první růžová část některé hrany dorazí k vrcholu, který nepatří do zvolené komponenty.

V této scéně je používána pouze Euklidovská metrika, protože okolí vrcholů zde nejsou dobře patrna a metriky $\mathcal{L}_1$ a $\mathcal{L}_{max}$ by uživatele spíš mátly.

Scéna: *Ohodnocený graf*

Tato scéna opakuje scénu předchozí, cena hrany je ale dána číslem zapsaným u hrany stejně jako v třetí scéně tohoto appletu.

Pro znázornění okolí se opět používají růžové segmenty. Jejich délka ale roste v různých hranách různou rychlostí, která je nepřímo úměrná ceně hrany. Znamená to, že délka růžového segmentu hrany s cenou c je t/c, kde hodnota t je stejná pro všechny hrany a roste s časem.

Nejbližší soused z jiné komponenty, nyní vzhledem k obecným cenám hran, je opět nalezen, když první růžová část některé hrany dorazí k vrcholu, který nepatří do zvolené komponenty.

Scéna: *Jarníkův - Primův algoritmus*

Tato scéna ukazuje algoritmus popsaný Jarníkem v roce 1930 a znovu objevený Primem v roce 1957, který je jednoduchou implementací obecného schématu. Na začátku se zvolí zcela libovolně vrchol v_0 a za komponentu $\mathcal{C}$ se v obecném schématu volí vždy komponenta obsahující vrchol v_0.

V průběhu výpočtu proto neustále narůstá jedna komponenta, zatímco ostatní zůstávají jednovrcholové a jsou "požírány" komponentou obsahující zvolený vrchol.

Scéna je plně funkční co do volby výchozího grafu i metriky a povolených hran. Existují tři základní volby: **Ohodnocený graf** přináší omezení v podobě povinných modrých hran s obecným číselným ohodnocením, **Metrický graf** má také povinné hrany, ale s Euklidovským ohodnocením a **Úplný metrický graf** nemá žádná omezení volby hran a jejich cena je určena metrikou. Je zde možno zvolit libovolnou z metrik $\mathcal{L}_2$, $\mathcal{L}_1$ a $\mathcal{L}_{max}$. Je také pochopitelně možno měnit množinu vrcholů a, pokud to přichází v úvahu, i hrany a jejich ceny. Odpadá zde pochopitelně volba mezi ruční a automatickou volbou komponenty; ta je dána jednoznačně algoritmem.

Efektivní implementace Jarník - Primova algoritmu vyžaduje vhodný způsob volby minimální hrany vycházející z narůstající komponenty obsahující vrchol v_0. Výhodně se dá provést pomocí haldy (viz kapitola 4) nebo podobné datové struktury, do které umístíme hrany vycházející z této komponenty s uvážením jejich ohodnocení. Když je do velké komponenty přidán nový vrchol v, hrany vedoucí z v do vrcholů, které již v komponentě jsou, se z prioritní fronty vynechají a hrany z v do vrcholů, které ještě v komponentě nejsou, se naopak přidají. Určení přidávané hrany se snadno provede pomocí operace určení minima, kterou halda implementuje.

Scéna: *Kruskalův algoritmus*

Kruskalův algoritmus na první pohled nespadá pod obecné schéma; je i poněkud odlišně vizualizován. Později však zdůvodníme, že i ten je implementací obecného schématu. Nyní ale si algoritmus vysvětlíme.

Kruskalův algoritmus je možno popsat takto: vychází se z grafu bez hran a dokud výpočetní les není souvislý (neboli dokud nemá $n-1$ hran, kde n je počet vrcholů), provádí se opakovaně provádí následující akce:

- mezi hranami spojující dvě *různé* komponenty výpočetního lesa se najde hrana s *minimální* cenou, a přidá se do výpočetního lesa.

V této scéně zkoumáme pro jednoduchost případ metrického grafu bez zakázaných hran s metrikou $\mathcal{L}_2$, $\mathcal{L}_1$ nebo $\mathcal{L}_{max}$. V naší animaci se dvojice vrcholů z různých komponent s minimální vzdáleností nalezne tak, že ze *všech* vrcholů všech komponent se začnou rozšiřovat okolí stejného s časem se zvětšujícího poloměru. (Tvar okolí je opět dán zvolenou metrikou). Okolí mají barvu odvozenou od barvy svých určujících vrcholů, což umožní snadno odlišit okolí dvou vrcholů z různých komponent.

Dotek nebo překrývání okolí vrcholů ze stejné komponenty ponecháváme bez povšimnutí, ale animace se zastaví v okamžiku, kdy se dotknou dvě okolí z různých komponent, tedy dvě okolí různé barvy, protože v tomto případě spojnice vrcholů v jejich středech je hledaná minimální spojnice dvou různých komponent. Místo doteku se označí.

Zkuste si výpočet podle Kruskalova algoritmu pro různé množiny vrcholů a metriky. Kruskalův algoritmus nedává uživateli žádnou volnost (s výjimkou případu, kdy existují dvě nebo více hran spojujících různé komponenty, které mají *stejné* minimální ceny - v tomto případě lze volit libovolnou z nich; Algovize by to provedla náhodně sama, aniž by se dotazovala uživatele).

I když to na první pohled tak nevypadá, i Kruskalův algoritmus je implementací výše uvedeného obecného schématu: krok volby komponenty se provede tak, že se nalezne spojnice h dvou různých komponent s minimální cenou a za komponentu $\mathcal{C}$ se zvolí komponenta obsahující jeden (libovolně vybraný) ze dvou konců hrany h. Jelikož h je hrana s minimální cenou mezi hranami spojujícími různé komponenty, je i hranou s minimální cenou mezi hranami spojujícími komponentu $\mathcal{C}$ s jinou komponentou a proto musí (nebo může - v případě existence více minimálních hran tohoto typu) být v následujícím kroku volby hrany obecným schématem zvolena jako propojení zvolené komponenty $\mathcal{C}$ s jinou komponentou.

Efektivní implementace Kruskalova algoritmu vyžaduje určení způsobu nalezení minimální hrany mezi komponentami. Touto otázkou se budeme blíže zabývat v následující scéně.

Scéna: *Kruskalův algoritmus ještě jednou*

Jak již bylo řečeno, minulá scéna ukazovala, jak nalézt minimální kostru v grafu, pokud umíme nalézt hranu minimální ceny, spojující dvě různé komponenty výpočetního lesa. Hledání v ní bylo prováděno animací, ze které sice bylo jasné, že výsledek je skutečně minimální hrana, nebylo ale zřejmé, jak animaci efektivně implementovat v počítači.

V této scéně ukážeme standardní a jednoduchý způsob, jak se hledání minimální hrany provádí. Na počátku výpočetní les neobsahuje žádné hrany. Výpočet začíná tím, že se hrany grafu setřídí do posloupnosti tak, aby jejich ceny neklesaly. Pak se hrany v tomto pořadí probírají a pro každou hranu se určí, zda by její přidání do výpočetního lesa vytvořilo cyklus. Pokud nevytvoří,

hrana se do výpočetního lesa přidá, jinak se vyhodí.

Tento postup zjevně do lesa vždy přidává tu z hran spojujících různé komponenty, která má nejmenší cenu. Hrany s menší cenou totiž již byly všechny probrány a buď měly oba konce ve stejné komponentě již když byly uvažovány a nebo byly do lesa přidány a od toho okamžiku již mají oba konce ve stejné komponentě také.

Otázkou je, jak určit, zda hrana má své konce v různých komponentách výpočetního lesa. Zde se výhodně použije faktorová množina, popisovaná v kapitole 6. Když je na začátku les bez hran, jeho komponenty jsou tvořeny jednotlivými vrcholy, což je výchozí stav faktorové množiny. Pro každou hranu se pak určí, do kterých komponent patří její konce. Patří-li do téže komponenty, hrana se vyhodí a faktorová množina se nemění. Patří-li do různých komponent, pak se hrana přidá do vytvářeného lesa, čímž se obě komponenty lesa spojí v jednu, což se také odrazí v tom, že je spojíme dohromady i ve faktorové množině.

V této scéně je možno zvolit všechny možnosti zadání: obecně ohodnocený graf i metrický graf se zakázanými hranami nebo úplný graf a v metrickém případě jsou dostupné všechny tři metriky $\mathcal{L}_2$, $\mathcal{L}_1$ a $\mathcal{L}_{max}$. Upozorňuji ale na to, že v případě úplného metrického grafu jsou hranami všechny (neorientované) dvojice různých vrcholů, kterých je $\binom{n}{2}$, kde n je počet vrcholů grafu a pro větší počet vrcholů je toto číslo hodně velké a krokování výpočtu může trvat velmi dlouho.

Pokud je na ovladači zvolen malý krok, probíraná hrana se nejprve označí červeně; pokud je přidána do výpočetního lesa, dostane v dalším kroku zpět svoji původní barvu a navíc zelený podklad, zatímco pokud by uzavírala cyklus tak zežloutne, aby byla vizuálně potlačena. Při delším kroku se zčervenání přeskakuje.

Nakonec poznamenávám, že v případě úplného metrického grafu, kdy by bylo třeba probrat všechny dvojice vrcholů a postup by byl dosti pomalý, se někdy může hledat minimální hrana jinak, s využitím geometrické informace, která je k dispozici. Podrobnější popis takových možností by ale byl mimo rámec této knihy.

Scéna: *Správnost*

V této scéně ukážeme, že obecné schéma dává *vždy* správné řešení, tedy minimální kostru zvoleného hranově ohodnoceného grafu. Klíčem je dokázat, že po celou dobu výpočtu platí následující podmínka:

Inv: existuje alespoň jedna minimální kostra grafu, která v sobě obsahuje všechny hrany výpočetního lesa (tedy hrany označené v appletu zelenou barvou).

Tvrzení (Inv) zjevně platí z triviálních důvodů na začátku výpočtu, kdy výpočetní les neobsahuje žádné hrany.

Pokud tvrzení (Inv) platí i na konci výpočtu, pak je výpočetní les minimální kostrou. Na konci výpočtu je totiž výpočetní les souvislý, obsahuje proto alespoň $n-1$ hran (n označuje počet vrcholů), ale zároveň je obsažen v nějaké minimální kostře $\mathcal{K}$, a ta má také $n-1$ hran. Proto výpočetní les musí být *roven* minimální kostře $\mathcal{K}$.

Stačí tedy ukázat, že přidání hrany, zvolené obecným schématem, do vytvářené kostry výše uvedenou podmínku (Inv) neporuší. Abychom ukázali, proč tomu tak je, přidáme do průběhu krokování vždy za volbu hrany několik dalších kroků, které sice vytvářenou kostru nemění a nic nového nepočítají, ale objasňují logické vazby v grafu.

Přidaná hrana zůstává po dobu přidaných kroků zbarvena červeně, tedy odlišně od starších hran ve vytvářené kostře (které jsou zelené) a budeme ukazovat, co by se stalo, kdyby její přidání porušilo podmínku (Inv), tedy pokud by žádná minimální kostra obsahující staré zelené hrany vytvářené kostry neobsahovala nově přidanou červenou hranu.

Také se dočasně ponechává původní barevné označení vrcholů komponent, které nově přidaná hrana spojuje. Jejich barvy se sjednotí až po provedení kroků důkazu.

Krok: *Označení zvolené komponenty*

Komponentu, která byla naposledy zvolena v kroku volby komponenty, budeme dále označovat jako $\mathcal{C}$ a její vrcholy zvýrazníme bílým podkladem. ◇

Krok: *Alternativní kostra*

Jelikož předpokládáme, že podmínka (Inv) platila, ale po přidání červené hrany neplatí, měla by existovat minimální kostra, obsahující všechny zelené hrany, ale ne hranu červenou.

V druhém vloženém kroku se vykreslí kostra, která obsahuje všechny starší (zelené) hrany vytvářené kostry, další hrany nakreslené žlutě, ale ne nově přidanou červenou hranu.

Ukážeme, že předpoklad, že kostra tvořená zelenými a žlutými hranami má menší cenu než libovolná kostra obsahující v daném okamžiku nový výpočetní les (tedy zelené hrany plus červenou hranu) vede ke sporu a tím dokážeme, že ve skutečnosti žádný krok obecného algoritmu nemůže podmínku (Inv) porušit.

◇

Krok: *Alternativní cesta*

Konce červené hrany musejí být v žluto-zelené kostře spojeny cestou; tuto cestu označíme π. Konce červené hrany pochopitelně nejsou spojeny přímo,

protože červená hrana do žluto-zelené kostry nepatří. V tomto kroku je cesta π znázorněna tak, že její hrany jsou obroubeny světle fialovou barvou. $\diamond$

Krok: *Nalezení můstku*

Jeden z koncových vrcholů červené hrany leží v komponentě $\mathcal{C}$, tedy je označen bílým podkladem a druhý konec červené hrany v $\mathcal{C}$ neleží. Barevný podklad hran původně světle fialově zvýrazněné cesty v žluté kostře, spojující konce červené hrany, se přebarví následujícím způsobem:
Barva první hrany, která má ještě jeden konec v $\mathcal{C}$, ale druhý již mimo $\mathcal{C}$, se změní na černou. Takováto hrana, které budeme říkat *můstek*, zjevně musí existovat. Pokud jsou v cestě hrany předcházející můstek, změní se navíc jejich barva ze světle na tmavě zelenou. Barva ostatních hran cesty se nezmění. $\diamond$

Krok: *Vynechání můstku*

Černý můstek i červená hrana jsou dvě hrany, které mají jeden konec v komponentě $\mathcal{C}$ (jejíž vrcholy jsou označeny bílým podkladovým kolečkem) a druhý konec mimo $\mathcal{C}$. Nově přidaná červená hrana byla volena tak, že má minimální ohodnocení mezi hranami spojujícími $\mathcal{C}$ s jinou komponentou. Proto musí být cena černého můstku *větší nebo rovna* ceně nově přidané červené hrany.

V tomto kroku se černý můstek ze žluto-zelené kostry vynechá a přidá se do ní červená hrana, čímž vznikne nová kostra grafu. Nově vzniklý soubor skutečně kostru vytváří: přidáním červené hrany se vytvoří jediný cykl, tvořený červenou hranou a hranami cesty π, který se ale přeruší vynecháním černého můstku. Vynechání černého můstku neporuší souvislost - co bylo spojeno přes černý můstek, je nyní propojeno červenou hranou spolu s hranami cesty π, které v kostře zůstaly.

Je jasné, že součet ohodnocení hran nové kostry s červenou hranou je buď menší nebo roven součtu ohodnocení hran původní žluté kostry s černým můstkem. V prvním případě tedy původní červená kostra nebyla minimální (nalezli jsme kostru, která je lepší) a v druhém případě jsme určili kostru, která je stejně dobrá jako původní žluto-zelená kostra, ale obsahuje červenou hranu. V obou případech dostáváme spor s původním předpokladem neplatnosti podmínky (Inv). $\diamond$

Kapitola 16

Toky v sítích

16.1 Hladový algoritmus

Cílem této části je definovat problém největšího toku v síti a zavést pojem rezervy hrany a přenosu přebytku toku, které pak umožní algoritmy pro hledání největšího toku formulovat jednoduše a přehledně. Popíšeme zde také jednoduchý algoritmus pro hledání toku, který je typu, obvykle nazývaného "hladový", a který sice neumožňuje nalézt optimální řešení problému, ale ukáže, proč je vhodné algoritmy formulovat v řeči rezerv a přenosu přebytku.

Síť je orientovaný graf spolu se dvěma jeho navzájem různými vrcholy, které označujeme jako *zdroj* (zobrazen zeleně) a *spotřebič* (zobrazen modře). Kromě toho je každé hraně grafu přiřazeno nezáporné reálné číslo, nazývané *kapacita* hrany. Vrcholy sítě různé od zdroje a spotřebiče nazýváme *vnitřní vrcholy*.

Tok v síti je funkce t, která každé hraně h přiřazuje číslo $t(h)$, pro které platí $0 \leq t(h) \leq c(h)$, kde $c(h)$ je kapacita hrany h a navíc pro každý vnitřní vrchol v platí, že součet toků hranami vstupujícími do v je roven součtu toků hranami vystupujícími z v, tedy

$$\sum_{h \in in(v)} t(h) = \sum_{k \in out(v)} t(k),$$

kde $in(v)$, resp. $out(v)$, je množina hran sítě vstupujících do v, resp. vystupujících z v.

Je možno si představit, že uzly jsou města, hrany jsou úseky železnice nebo silnice, kterými se snažíme dopravovat jistý druh zboží z jednoho města (zdroj) do jiného města (spotřebič), přičemž každá cesta má omezenou průchodnost a zboží se skutečně jen přepravuje, ale v žádném jiném než výchozím nebo cílovém městě se neskladuje, nespotřebovává ani neztrácí, ale také nevzniká.

Je také možno si hrany představovat jako vodovodní nebo ropná potrubí nebo elektrická vedení, kdy je požadavek bezeztrátové dopravy obvykle přirozeným způsobem splněn.

V rámci výše uvedených omezení se budeme snažit o přepravu co největšího množství zboží nebo tekutiny ze zdroje. Toto množství je dáno přebytkem toku ve spotřebiči (kam obvykle tok pouze přitéká, i když definice nezabraňuje tomu, aby část byla současně odváděna a třeba i vedena zpět do zdroje) a je současně rovno deficitu zdroje (ze kterého obvykle tok jen odtéká).

Z důvodů, které uvidíme později, budeme předpokládat, že pokud (v, w) je hrana sítě, pak i hrana opačná, t.j. (w, v) do sítě patří. Pokud by v síti nebyla, tak ji přidáme s nulovou kapacitou (což je z hlediska toků totéž jako kdyby tam nebyla).

Scéna: *Největší tok*

Tato scéna umožňuje si pro zobrazenou síť nechat ukázat největší tok stisknutím knoflíku **Krok** (a vrátit se do výchozího stavu knoflíkem **Zpět).** Před provedením výpočtu toku je v této scéně u každé hrany uvedena její kapacita; po výpočtu toku je u ní uvedena dvojice tok:kapacita (a hrany, kterými teče nulový tok zešednou). Tak tomu je, pokud je na ovladači zvoleno **Automatické označení.** Změnou volby je možno dosáhnout toho, že je stále zobrazena kapacita nebo stále zobrazen tok a nebo je stále zobrazena dvojice čísel tok:kapacita.

Ověřte si, že pro vypočtený tok platí výše uvedené podmínky. Ověření optimality je také možné, ale není jednoduché. Je možné jej provést způsoby, které budou uvedeny později.

Scéna umožňuje běžné úpravy sítě jako přidávání, odebírání a přesouvání vrcholů a přidávání a odebírání hran a změny jejich kapacity (po klepnutí na hranu pravým knoflíkem myši a volbě v popup menu). Je také možno změnit zdroj i spotřebič (po klepnutí na zvolený vrchol pravým knoflíkem myši vybrat v popup menu) nebo zobrazit některý z předdefinovaných příkladů. K editaci je třeba volbu **Počítej** změnit na **Edituj.**

Scéna: *Hladový algoritmus*

Základní myšlenka Ford-Fulkersonova algoritmu je jednoduchá: začínáme nulovým tokem (žádnou hranou nic neteče), který postupně zvětšujeme tak dlouho, dokud nenalezneme největší možný tok. Nejjednodušší (ale, jak ukážeme dále, nedostatečný) způsob zvětšení toku je následující:
nalezneme orientovanou cestu ze zdroje do spotřebiče takovou, že každou její hranou protéká méně toku než je její kapacita, takže je možné tok ještě o něco zvýšit, a pak zvýšíme hodnotu toku všemi hranami o jistou hodnotu Δ.

Hranami cesty musí protékat tok menší než jejich kapacity, aby tok bylo možno zvětšit bez porušení kapacitního omezení. Proto tok hranou h s kapacitou $c(h)$, kterou již teče tok $t(h)$, je možné zvýšit nejvýše o $c(h) - t(h)$. V

důsledku toho největší hodnota Δ, která neporuší kapacitní omezení, je rovna $\min_h(c(h) - t(h))$, kde minimum se bere přes hrany cesty. Kirchhofův zákon (rovnost přítoku a odtoku ve všech vnitřních vrcholech sítě) se neporuší, protože pro každý vrchol cesty různý od zdroje a spotřebiče se přítok i odtok zvýší o Δ. Velikost toku vycházejícího ze zdroje se ovšem zvýší o Δ.

Krokujte si algoritmus a sledujte hledání a zpracování cest. Na ovladači je možno nastavit, jak detailně se krokování bude provádět.

Uvedený algoritmus se obvykle nazývá hladový, protože jen stále ujídá kapacitu hran pro zvýšení toku a není ochoten tok některou hranou snížit pro dosažení globálního zlepšení.

Výpočet probíhá pomalu, ale pro pochopení myšlenky algoritmu stačí jej sledovat jen chvíli.

Bohužel se může stát, jak uvidíme v příští scéně, že další zlepšení toku využitím přímé zlepšující cesty není možné, ale optimální tok přesto není nalezen.

V této a mnoha následujících scénách se také ukazuje pseudokód algoritmu, ve kterém je znázorněno, jak výpočet probíhá. Pokud vám pseudokód vadí, vypněte jeho zobrazování (změňte **Program automaticky** na **Skryj program).**

Scéna: *Hladový algoritmus - selhání*

V této scéně je ukázána jedna velmi jednoduchá síť, ve které postupem podle předchozí scény nalezneme tok, který již hladový algoritmus dále zlepšit neumí, ale *není* největší možný, neboť v síti existuje odlišný tok větší velikosti. Je z toho vidět, že neustálé zvyšování toku použitím přímé zlepšující cesty může vést do slepé uličky a, jak uvidíme dále, je občas nutno přikročit k tomu, že se tok některými hranami třeba i opakovaně sníží a pak se začne zvyšovat jiným způsobem a teprve tak bude možno nalézt optimální tok.

U zobrazeného příkladu optimální tok využívá plné kapacity hran na obvodu sítě, ale nevyužívá příčku přes střed sítě; jeho velikost je 2. Výpočetní postup je volen tak, že ale hladový algoritmus nalezne tok velikosti 1, který již nelze zlepšit pouhým zvětšováním toků hranami. Je to proto, že při zpracování první nalezené cesty se zvýší tok příčkou na 1, ale snadno si můžete ověřit, že žádný tok, ve kterém příčkou teče nenulová hodnota toku, nemůže mít optimální velikost 2. Přitom hladový algoritmus neumí tok příčkou zmenšit.

Volbu zlepšujících cest provádí Algovize sama, protože je možno ukázat, že v libovolné síti lze hladový algoritmus navádět (t.j volit cesty pro zlepšení toku) takovým způsobem, že optimální tok najde. Například v našem příkladu by stačilo napřed vybrat cestu složenou ze dvou horních hran a pak cestu obsahující spodní dvě hrany. Mohlo by se proto stát, že uživatel bude záměrně nebo náhodou volit cesty tak, že nalezne optimální tok, zatímco cílem scény

je ukázat, že algoritmus může zabloudit a zastavit se, aniž by největší možný tok nalezl.

Scéna: *Přenos přebytku*

V následujících několika scénách bude ukázán jiný pohled na zlepšování toku, který umožní upravit hladový algoritmus tak, aby bylo optimální řešení vždy nalezeno. Toto zlepšení popsali v roce 1964 Ford s Fulkersonem, po nichž je zlepšený algoritmus nazván.

Představme si, že tok hranami cesty zvětšujeme o určenou hodnotu Δ postupně v tom pořadí, v jakém se na cestě nacházejí. Pak je v průběhu této akce vždy v jednom vrcholu (koncovém vrcholu naposledy zpracované hrany) porušena podmínka rovnosti toku přitékajícího a toku odtékajícího - do vrcholu přitéká o Δ jednotek toku více než před zpracováním cesty, ale odtéká zatím stejně. Tento rozdíl budeme nazývat přebytek.

Využívajíce analogii s vodovodní sítí budeme přebytek znázorňovat výší pruhovaného sloupečku nad vrcholem, představujícího vyrovnávací nádrž s proměnlivou výškou hladiny, který je úměrná rozdílu přítoku do vrcholu a odtoku z vrcholu. Jestliže je přebytek záporný (tedy ve skutečnosti deficit - nastává to pouze u zdroje), bude sloupeček pod vrcholem představovat hladinu ve studni, ze které rozdíl dočerpáváme.

Zkuste si knoflíkem **Krok** krokovat průběh výpočtu zpracování první cesty. Algovize nejprve najde (hranu po hraně) cestu ze zdroje do spotřebiče a určí patřičné Δ. Potom, opět po jednotlivých hranách, zvyšuje tok. Hrana, ve které se bude tok zvyšovat, je znázorněna červeně; je vidět, že zvýšení toku hranou má za následek odvedení přebytku z jejího počátečního vrcholu do jejího koncového vrcholu. V okamžiku, kdy se kladný přebytek dostane až do spotřebiče (ve kterém je přípustný), zpracování cesty končí a začíná určení a zpracování další cesty (pokud existuje).

Algoritmus pracuje podobně jako v předchozí scéně - zvolí si nejprve cestu přes příčku, takže pokud by pracoval hladově, po zpracování první cesty by se zastavil, aniž by byl nalezen optimální tok. Zde ale výpočet pokračuje dále. Místo aby nalezl další orientovanou cestu ze zdroje do spotřebiče, Algovize ukáže postupně tři hrany, z nichž prostřední je ale v protisměru.

Pak se zahájí přenos přebytku ze zdroje do spotřebiče. Nejprve se již známým způsobem odvede další jednotka toku za zdroje do spodního vrcholu zvýšením toku příslušnou hranou. Pak se střední příčka zobrazí jako široká šipka; její šířka odpovídá kapacitě a šířka červeného pruhu odpovídá toku, který jí prochází.

Jelikož v tomto okamžiku širokou hranou prochází tok velikosti 1, který je roven její kapacitě, je červený pruh roztažen přes celou její šířku. Přejeďte ale myší se stisknutým knoflíkem přes hranu. Hrana červeného pásu sleduje kurzor, takže se jeho šířka mění a tím je možno měnit velikost toku hranou z

maximální hodnoty až na nulu.

Snižování toku příčkou (tok teče shora dolů) způsobí, že se přebytek ve spodním vrcholu příčky snižuje a o tutéž hodnotu se ale zvyšuje přebytek horního vrcholu příčky (původně nulový). Přebytek je tedy také možno převést z jednoho vrcholu do druhého tak, že se *sníží tok hranou v protisměru*, pokud jí teče nenulový tok.

V původní verzi proto Ford-Fulkersonův algoritmus zlepšoval tok podle tak zvaných zobecněných cest, na kterých byly hrany ve směru s rezervou a hrany v protisměru s nenulovým tokem. Než ukážeme, že takto sestrojený algoritmus již *vždy* nalezne optimální tok, zavedeme si v následujících několika scénách pojmy, které umožňují formulovat algoritmus jednotným a zjednodušeným způsobem, bez neustálého rozdělování hran zobecněné cesty na přímé a zpětné.

Scéna: *Tok hranou*

Abychom nemuseli stále rozebírat, zda k přenosu přebytku dochází hladovým zvyšování toku hranou nebo snižováním toku protisměrnou hranou, zavedeme v několika následujících jednoduchých scénách způsob práce s přebytky, který situaci zjednoduší.

Tato scéna je jen přípravná a graficky znázorňuje toky ve dvou opačně orientovaných hranách mezi dvěma vrcholy. Táhnutím myši uvnitř jedné z těchto širokých hran se mění šířka pásu, který graficky znázorňuje velikost toku hranou (v horní hraně, jdoucí zleva doprava, je světle zelený, v dolní hraně směřující vlevo, je tmavě zelený). Šířka hrany odpovídá její kapacitě. Zkuste v obou hranách nastavit nenulový tok. Žluté sloupce u konců hran znázorňují přebytky toku v koncových vrcholech hran (nebo přesněji příspěvky k přebytkům toku od zobrazených hran). Přebytek je kladný, pokud je sloupec nad dělící čárou mezi vyobrazenými hranami, záporný pokud je sloupec pod čárou. Měňte toky a sledujte, že vždy jeden přebytek roste a druhý o tutéž hodnotu klesá.

Scéna: *Cirkulace*

Scéna je v zásadě opakováním předchozí. Obecná definice i náš applet připouští, aby oběma opačnými hranami tekl kladný tok. Z praktického hlediska i z hlediska hledání největšího toku to ale není příliš účelné; vozíme-li jisté zboží z Prahy do Brna a současně stejné zboží vezeme z Brna do Prahy, jen plýtváme prostředky.

Ta část toku, která teče v obou hranách současně, je na obrazovce znázorněna modře a nazýváme ji cirkulace. (Pro sečtělé čtenáře: náš pojem cirkulace je jen speciálním případem toho, co se obvykle cirkulací také nazývá).

Jestliže cirkulaci odečteme od toků oběma hranami, naše situace je může jen zlepšit: Toky se sníží při zachování jejich nezápornosti, takže kapacitní

omezení zůstanou v platnosti, ale hrany mají více rezervy pro další zvyšování toku. Současné snížení toku v obou hranách o stejnou hodnotu také nezmění přebytky ve vrcholech a proto tok zůstane tokem (nulové přebytky ve vnitřních vrcholech sítě) a jeho velikost (přebytek ve spotřebiči) se také nezmění. Z toho plyne, že existuje maximální tok (náš cíl), bez cirkulací.

Scéna: *Tok bez cirkulace*

Tato scéna se podobá předchozí, ale ukazuje již toky bez cirkulací. Jestliže tedy v jedné z obou hran je nenulový tok, tok druhou je nulový. Zkuste si opět nastavovat toky hranami; požadavek nenulového toku jednou hranou vynuluje tok v druhé z nich.

Se stisknutou myší se nyní pohybujte od horního okraje horní hrany ke spodnímu okraji dolní hrany. Je zřejmé, že okamžitý stav (toky hranami) je popsán jediným parametrem - výškou rozhraní taženého myší - který se mění v rozmezí od vrchního okraje horní hrany po spodní okraj dolní hrany a někdy představuje tok zleva doprava, jindy tok směrem opačným. Jak rozhraní stoupá, zvyšuje se přebytek v pravém vrcholu a současně klesá přebytek levého vrcholu.

Je snadno vidět, že rezerva pro zvýšení přebytku vrcholu vpravo (a současně pro snížení přebytku vlevo) je dána vzdáleností horního okraje horní hrany a pohyblivého rozhraní, jak je naznačeno červenou kótou s popisem "rezerva" umístěným nad hranami. Pokud je tok horní hranou kladný, pak tato rezerva je rovna její kapacitě, snížené o velikost toku hranou; rezerva je tudíž menší než její kapacita. Jestliže ale je kladný tok spodní hranou, který jde v opačném směru, pak přebytek zleva doprava lze přenášet tak, že se nejprve snižuje až na nulu tok spodní hranou a pak se zvyšuje tok horní hranou až do její kapacity. Rezerva pro přenos přebytku je tedy *větší* než kapacita horní hrany. Obdobně tomu je i se spodní hranou.

Scéna: *Rezerva*

Tato scéna opakuje předchozí, ale je v ní vynechána hranice mezi hranami, aby bylo zdůrazněno, že možnost zvyšování nebo snižování přebytků vrcholů nezávisí na této hranici, která určuje kapacity hran, ale pouze na vzdálenosti mezi horním okrajem horní hrany a dolním okrajem dolní hrany, který je dán součtem kapacit těchto hran. Rezerva pro zvýšení přebytku pravého vrcholu, daná vzdáleností horního okraje horní hrany a pohyblivého rozhraní (šířka bílého pásu) a rezerva pro zvýšení přebytku levého vrcholu, daná vzdáleností pohyblivého rozhraní a dolního okraje dolní hrany (šířka šedého pásu), jsou pak určeny polohou pohyblivého rozhraní. Součet těchto rezerv pro zvýšení přebytku je zjevně konstantní a rovný součtu kapacit hran.

Pro úplný popis situace proto bude vždy (předpokládajíce znalost kapacit hran) dostatečné znát rezervu jedné z hran (druhou lze snadno dopočítat).

Scéna: *Tok a rezerva*

Pro popis algoritmů je daleko výhodnější používat rezervy hran a ne jejich toky, protože přenos přebytku podél hrany e se v řeči toků musí rozdělovat do dvou možností: zvýšení toku hranou e nebo snížení toku hranou opačnou, zatímco v řeči rezerv jde v obou případech o snížení rezervy hrany e (za současného zvýšení rezervy hrany opačné o stejnou hodnotu, protože součet těchto rezerv je konstantní a rovný součtu kapacit hran).

Za předpokladu nulového výchozího toku se proto na počátku výpočtu položí výchozí rezervy hran rovné jejich kapacitám a tok se upravuje tak, že se přenáší podél hran přebytky ve vrcholech za současné změny rezerv dotčených hran. Na konci výpočtu ale je nutné zpět z rezerv hran spočítat toky jednotlivými hranami. Tato scéna ukazuje, jak se to provede.

Nastavte pomocí myši vhodně rezervy jako v předchozí scéně (t.j. zvolte polohu rozhraní mezi bílým pásem, který je rezervou pro přenos zleva doprava a šedým pásem - rezerva zprava doleva) a klikněte na checkbox **Výpočet toku.** Absolutní velikost toku se zobrazí jako šířka zeleného pásu a směr toku, t.j. jedná-li se o tok hranou zleva doprava nebo zprava doleva, se pozná podle zkosení daného čelem odpovídající hrany.

Scéna: *Nulová kapacita*

Nyní ještě závěrečná scéna týkající se toků a kapacit: spodní hrana zde má nulovou kapacitu (což je vlastně totéž jako by v síti nebyla). Z obrázku je ale vidět, že tato hrana může i přesto mít nenulovou kladnou rezervu; ta je v takovém případě rovna toku opačnou hranou a znamená, že změna přebytků koncových vrcholů se dá provést snížením toku opačnou hranou.

V následujících scénách týkajících se toků v sítích budeme pro formulaci algoritmů používat výhradně pojmu rezervy hrany a toky hranami se budou dopočítávat až dodatečně na konci výpočtu.

Ve většině scén budeme používat zobrazení ukazující ty hrany, které mají kladnou rezervu (včetně těch, které byly do sítě doplněny dodatečně s nulovou kapacitou), ale na druhé straně *skrývající* hrany s nulovou rezervou, protože ty již nejsou použitelné k dalšímu přenosu přebytku. (Týká se volby **rezerva** a **rezerva:kapacita** - v některých scénách a v módu **Práce** lze způsob zobrazení volit z více možností).

16.2 Ford-Fulkersonův algoritmus

Nyní shrneme a trochu rozvineme to, co bylo vysvětleno v předchozí subkapitole. Předpokládejme, že je na hranách sítě definována jistá funkce t, která sice vyhovuje stejným kapacitním omezením jako tok, ale může mít nezáporný přebytek i ve vnitřních vrcholech (tedy je povoleno, aby do vnitřních vrcholů při-

tékalo hranami více než odtéká). Takové funkci budeme říkat *vlna*[1]. Speciálně tedy t může být tok, tak jak byl definován v první scéně. Pak řekneme, že rezerva hrany $h = (u, v)$ je číslo $r(h)$, definované jako $r(h) = c(h) - t(h) + t(h^{op})$, kde h^{op} představuje hranu (v, u). Rozpomeňte se na naši úmluvu, že s každou hranou $h = (u, v)$ je v síti i hrana opačná $t(h^{op}) = (v, u)$, ale že nebudeme připouštět, aby platilo současně $t(h) > 0$ a $t(h^{op}) > 0$. V definici rezervy proto je vždy jeden ze sčítanců nulový.

Pokud má hrana $h = (u, v)$ kladnou rezervu R a pokud t je vlna, která má ve vrcholu u kladný přebytek E, pak pro libovolné kladné číslo Δ takové, že $\Delta \leq min(R, E)$, je možno manipulací s hodnotami t pro hranu $h = (u, v)$ a/nebo $h^{op} = (v, u)$ (pokud jsou hranami sítě) dosáhnout snížení přebytku t ve vrcholu u o hodnotu Δ za současného zvýšení přebytku vrcholu v o tutéž hodnotu Δ. Přitom dojde současně k tomu, že rezerva hrany h o Δ poklesne a současně se rezerva opačné hrany h^{op} o Δ zvýší. Této operaci budeme říkat přenesení přebytku velikosti Δ z vrcholu u do vrcholu v.

Konkrétně manipulace s toky v h a/nebo h^{op} při přenášení přebytku z vrcholu u do vrcholu v vypadá takto:

- jestliže $0 < \Delta \leq t(h^{op})$, pak hodnotu $t(h^{op})$ snížíme o Δ;
- jestliže $0 < t(h^{op}) < \Delta$, pak hodnotu $t(h)$ zvýšíme o $\Delta - t(h^{op})$ a pak hodnotu $t(h^{op})$ snížíme na nulu;
- jestliže $t(h^{op}) = 0$, pak hodnotu $t(h)$ zvýšíme o Δ;

Nakonec si povšimněte, že pokud t je nulový tok, pak rezervy hran sítě jsou rovny jejich kapacitám.

Hranám, které mají kladnou rezervu, budeme také říkat *nenasycené* a hranám s nulovou rezervou *nasycené*. Cesta složená z nenasycených hran se bude nazývat *nenasycená* cesta.

Scéna: *Ford-Fulkersonův algoritmus*

Jak už bylo naznačeno výše, Ford-Fulkersonův algoritmus je s využitím pojmu rezervy hrany možno popsat velmi jednoduše. Místo práce v původní síti budeme pracovat v síti, kterou budeme nazývat síť rezerv. Síť rezerv má stejné vrcholy jako původní síť, ale množina jejích hran závisí na tom, jaký v základní síti teče tok nebo obecněji vlna a proto se síť rezerv v průběhu výpočtu stále mění. Předpokládáme-li, že v jistý okamžik teče základní sítí vlna t, pak hranami v síti rezerv právě ty hrany $h = (u, v)$ původní sítě, které mají kladnou rezervu, tedy pro které platí $r(h) > 0$.

[1] V angličtině se tok nazývá *flow* a vlna je *pre-flow*. Česky by se tedy mohly pro vlnu použít výrazy jako "předtok" nebo "pratok", ale jejich ošklivost mne vedla k tomu, že jsem vynalezl zcela odlišné pojmenování, které ale podle mého dobře vystihuje, jak se "pre-flow" používá zde a především u Goldbergova algoritmu

Algoritmus začíná se z nulového toku a dokud je možné nalézt cestu ze zdroje do spotřebiče, která je tvořena hranami s kladnými rezervami, pak jednu z takových cest vybereme a po jejích hranách přesuneme ze zdroje do spotřebiče přebytek rovný minimu Δ z rezerv hran cesty. Tato hodnota je největší možná velikost přebytku, který se dá postupně hranami přenést, aniž by došlo k porušení kapacitních omezení. Je zřejmé, že touto operací se velikost toku zvýší o Δ.

Zkuste si výpočet pro zobrazený příklad a případně i další sítě z nabízeného souboru nebo vlastní konstrukce. Na obrazovce je (s výjimkou začátku a konce výpočtu) zobrazována síť rezerv, tedy šipky představují hrany, které mají kladnou rezervu. Je velmi důležité, abyste pochopili, jak se při zpracování jedné cesty změní rezervy hran a jak se změní množina hran sítě rezerv: všem hranám zpracovávané cesty poklesne rezerva o Δ. Alespoň jedné hraně cesty (ale často i více, například všem hranám cesty) poklesne rezerva na nulu a tím pádem ze sítě rezerv vypadne. Zároveň se všem opačným hranám rezerva o Δ zvýší a proto se všechny v síti rezerv objeví, pokud v ní již předtím nebyly.

Scéna: *Řez a optimalita*

Předchozí scéna ukázala, jak Ford-Fulkersonův algoritmus zvyšuje tok pomocí zlepšujících cest. Na první pohled ovšem není jasné, zda v okamžiku, kdy algoritmus nenachází další zlepšující cestu, je nalezený tok největší možný. V této scéně ukážeme, že tomu tak je.

Pro zvolenou síť proveďte celý výpočet Ford-Fulkersonova algoritmu. Volba kroku je nastavena na nejvyšší hodnotu **Celý výpočet,** takže je to možné provést jediným stisknutím knoflíku **Krok.** Můžete ale také volbu kroku změnit a výpočet provádět po větších či menších krocích postupně.

Řekneme, že vrchol sítě je dosažitelný, pokud do něho vede cesta složená z hran s kladnými rezervami. Zdroj je z triviálních důvodů dosažitelný, ale spotřebič v okamžiku zastavení výpočtu dosažitelný není, protože by se jinak Ford-Fulkersonův algoritmus nezastavil, ale upravoval by tok dále podle některé cesty, kterou lze spotřebič dosáhnout ze zdroje.

Provádějte výpočet až do jeho ukončení. V tento okamžik se barevné označení vrcholů změní. Všechny dosažitelné vrcholy se zbarví na stejnou barvu jako zdroj, tedy na zelenou. Všechny nedosažitelné vrcholy se zbarví na stejnou barvu jako spotřebič, tedy modrou. Dojde k rozdělení množiny vrcholů do dvou částí - dosažené a nedosažené, a toto rozdělení nazveme *řez*.

Změní se také barvy hran. Mějte na paměti, že nejsou zobrazeny všechny hrany, ale právě ty hrany, které mají kladnou rezervu. Z nich hrany spojující dva zelené dosažitelné vrcholy se také zbarví zeleně a hrany spojující dva modré nedosažitelné vrcholy se zbarví modře. Nakonec hrany vycházející z nedosažitelného vrcholu a končící ve vrcholu dosažitelném jsou obarveny červeně.

Povšimněte si, že nevidíme žádnou hranu vedoucí z nějakého zeleného (tedy dosažitelného) vrcholu u do nějakého modrého (tedy nedosažitelného) vrcholu v, protože kdyby byla zobrazena, měla by kladnou rezervu a orientovaná cesta v síti rezerv ze zdroje do u (musí existovat, protože u je dosažitelný) spolu s hranou (u, v) by znamenala, že i v by byl dosažitelný, což by byl spor s naším výchozím předpokladem.

Nyní je myslím zcela jasné, že ze zelené části sítě se do modré části nedá žádným způsobem dopravit více toku, protože žádná z hran, které jsou v tomto směru, již nemá použitelnou rezervu. Matematicky korektní důkaz tohoto tvrzení však zde podávat nebudu, odkazuji vás na standardní učebnice grafových algoritmů.

Scéna: *Rychlost výpočtu*

V předchozí scéně jsme ukázali, že Ford-Fulkersonův algoritmus na rozdíl od hladového algoritmu vždy nalezne optimální řešení.

Jedinou otázkou je, jak dlouho to trvá. Myslím, že nebudete mít trpělivost dokončit výpočet, který bude probíhat na obrazovce, ale jeho myšlenku jistě pochopíte. Asi si všimnete, že Algovize cesty volí záměrně tak, aby výpočet nesměřoval k cíli příliš rychle. Mějte však na paměti, že Ford-Fulkersonův algoritmus nepředepisuje volbu cesty, pokud je více možností a proto je tento postup zcela legitimní. V případě iracionálních kapacit se dokonce může stát, že výpočet neskončí nikdy.

Zkuste si výpočet krokovat - uvidíte, že zpracování každé cesty zlepší tok o 1. Liché iterace naleznou cestu prochází ze zdroje přes horní vrchol do dolního vrcholu a pak do spotřebiče, což vede ke zvýšení toku v příčce na 1. Sudé iterace pak ze zdroje jdou nejprve do dolního vrcholu, protisměrně příčkou do horního vrcholu a pak do spotřebiče, což vede ke snížení toku příčkou na nulu a hra se může opakovat. Jelikož, jak jistě vidíte, je velikost maximálního toku v zobrazené síti 2000, trval by výpočet opravdu dlouho; jsem přesvědčen, že neprojdete ani prvních padesát cest, které Algovize nabídne.

Počítač by sice pro tuto síť výpočet dokončil v rozumné době, ale již v rámci 32-bitových celých čísel by bylo možno změnit kapacity hran na obvodu na zhruba 2 miliardy a tedy by počet iterací dosáhl okolo 4000000000 a použitím 64-bitových hodnot kapacit by se doba trvání výpočtu mohla zvýšit na hodnotu, přesahující současné (a zřejmě i budoucí) výpočetní možnosti lidstva. Je proto žádoucí nalézt rychlejší algoritmus, kde by především bylo možné stanovit horní odhad doby výpočtu jen ze znalosti počtu hran a vrcholů a nebylo by možno výpočet libovolně prodlužovat změnou kapacit. Dva takové algoritmy, Dinitzův a Goldbergův, budou popsány v další části.

16.3 Dinitzův algoritmus

Scéna: *Dinitzův algoritmus*

Dinitzův algoritmus je vlastně implementací Ford-Fulkersonova algoritmu. Ford-Fulkersonův algoritmus nepředepisuje, jaká zlepšující cesta (t.j. nenasycená cesta ze zdroje do spotřebiče) se má vybrat, pokud je jich k dispozici více. Dinitzův algoritmus stanoví, že je třeba vybrat tu z nich, která obsahuje nejméně hran. Jak uvidíme, tento jednoduchý trik způsobí překvapivou změnu nejhoršího možného chování algoritmu. Nezávisle jej objevili také Edmonds a Karp, ale Dinitzova implementace navíc obsahuje v každé své fázi předzpracování, které výpočet ještě urychlí.

V hrubých krocích je výpočet ukázán na obrazovce. Provádějte výpočet krokovacími knoflíky; jednotlivé fáze výpočtu zde budou komentovány. Vypínatelným způsobem Algovize také ukazuje schéma programu, který algoritmus implementuje a znázorňuje v něm postup výpočtu. Bloky schématu korespondují s kroky výpočtu, jak jsou popsány dále. Až na výjimky je na obrazovce ukázána síť rezerv, tedy jsou nakresleny jen ty hrany, které mají rezervu nenulovou.

Krok: *Inicializace*

Zde se provedou jednoduché úvodní operace, především určení počátečních rezerv hran, které nebudu blíže komentovat. ◇

Krok: *Určení vrstev*

Vrcholy rozdělíme do vrstev; vrchol v je v k-té vrstvě, pokud cesta s nejmenším počtem hran ze zdroje do v, tvořená hranami s kladnou rezervou, obsahuje k hran. V nulté vrstvě je tedy jen zdroj, v první vrstvě vrcholy, do kterých vede ze zdroje hrana s kladnou rezervou, v další vrstvě vrcholy, do kterých vedou hrany s kladnou rezervou z první vrstvy atd. Vrstvy jsou naznačeny svislými pásy.

Obecně se může stát, že do některého vrcholu v síti rezerv cesta ze zdroje vůbec nevede. Takové vrcholy by byly zobrazeny u pravého okraje obrazovky.

Je vám jistě zřejmé, že rozdělení do vrstev se hledá prohledáním grafu rezerv do šířky, viz Kapitola 13.

Když se vrcholy dělí do vrstev poprvé, jsou toky hranami rovny nule a tedy jejich rezervy jsou rovny kapacitám, ale rozdělování do vrstev se bude provádět opakovaně a později již bude vazba mezi rezervami a kapacitami minimální a rozdělení do vrstev může být zcela odlišné.

V této fázi přiřadíme každému vrcholu v číslo $L(v)$, které říká, kolik hran má nejkratší cesta z počátku do v v síti rezerv (tedy cesta složená z hran s kladnými rezervami). Pokud vrchol v není v síti rezerv dosažitelný z počátku, pak $L(v) = \infty$. ◇

Krok: *Test ukončení výpočtu*

Pokud by v síti nevedla ze zdroje do spotřebiče cesta tvořená nenasycenými hranami, což bychom při určování vrstev zjistili tím, že $L(s) = \infty$ (kde s je spotřebič), pak výpočet končí vyskočením z cyklu a provedením závěrečných úprav; jak jsme již ukázali dříve, znamená to, že již byl nalezen maximální tok. ◇

Krok: *Vynechání pomalých hran*

Do tohoto kroku se dostaneme, pokud při určování vrstev bylo mimo jiné zjištěno, že v síti rezerv existuje cesta ze zdroje z do spotřebiče s. Konečné $L(s)$ pak udává délku této cesty. Dinitzův algoritmus přikazuje, aby pro zlepšování toku byla v síti rezerv vybrána ta cesta ze zdroje do spotřebiče, které má minimální délku, tedy délku rovnou $L(s)$.

Pozorováním výpočtu se zjistí, že většinou takových cest bývá více, někdy i velké množství a řadu z nich postupně vybereme pro zlepšení toku. Proto je užitečné okamžitou situaci prozkoumat podrobněji a provést předběžné úpravy, díky nimž bude snazší tyto cesty hledat. Toto předzpracování, které schází v Edmonds-Karpově algoritmu, způsobí, že Dinitzův algoritmus má lepší asymptotický odhad nejhoršího možného chování.

Podíváme-li se na graf na obrazovce, je zřejmé, že nasycená cesta ze zdroje do spotřebiče (tedy cesta skládající se z viditelných hran) a obsahující minimální počet hran se nemůže nikde zdržovat a každou hranou musí postoupit o jednu vrstvu doprava. Takové hrany budeme nazývat *rychlé*. Hraně, která má oba konce ve stejné vrstvě a nebo dokonce vede směrem vlevo (její konec je zdroji blíže než její počátek), které budeme říkat *pomalá*. Takovou hranou nejkratší nenasycená cesta procházet nemůže.

Stejně tak nemůže cesta minimální délky procházet vrcholy, které mají vzdálenost od zdroje větší nebo rovnou vzdálenosti spotřebiče od zdroje (a jsou různé od spotřebiče). Hrany vycházející z takových vrcholů také označíme za pomalé.

Pomalé hrany se poznají snadno na základě hodnot L. Má-li hrana počátek v a konec w, pak je pomalá právě když $L(w) \leq L(v)$ nebo $L(v) \geq L(s)$, kde s je spotřebič.

Na obrazovce se v tomto kroku pomalé hrany označí žlutě, rychlé hrany zůstávají modré.

Žádnou pomalou hranou (nyní je žlutá), nemůže procházet nejkratší nenasycená cesta ze zdroje do spotřebiče. Naopak to ale zjevně neplatí: leckterá modrá hrana je taková, že přes ni také taková cesta nevede. Takových hran se ale zbavíme později.

Tento krok je natolik jednoduchý, že jej již podrobněji rozebírat nebudu.

Zatržení checkboxu **Skryj žluté** se žluté pomalé hrany přestanou zobrazovat vůbec. Zobrazování výpočtu je pak názornější, ale zakrývá fakt, že v síti mají kladnou rezervu i pomalé hrany. ◇

Krok: *Nalezení slepých konců*

Jestliže z vrcholu vycházejí jen pomalé hrany, pak přes něj určitě nevede nejkratší nenasycená cesta. Totéž platí i pokud do vrcholu vstupují jen pomalé hrany. Takové vrcholy nyní určíme. Tato jednoduchá operace bude později rozebrána podrobněji, ale jistě by vám nečinilo problém příslušný podprogram napsat sami. ◇

Krok: *Pročištění sítě*

Jak jsme již viděli v předchozím kroku, přes některé rychlé (modré) hrany nevede žádná nejkratší nenasycená cesta ze zdroje do spotřebiče, protože pokud přes ně přejdeme, dostaneme se dříve či později do slepé uličky, neboli do vrcholu (jiného než spotřebič), ze kterého žádné rychlé modré hrany nevystupují.

Podobně je tomu i u hran, do kterých se ze zdroje nemůžeme dostat rychlými hranami.

V tomto kroku se všechny slepé uličky odstraní - hrany, které jsou sice rychlé, ale vedou do slepé uličky nebo se do nich dostaneme jen ze slepé uličky a ne ze zdroje. Jak se to provede ukážeme později, v této scéně uvidíme jen výsledek a připomeneme, že v některých případech může být slepá ulička dlouhá a až za dlouhou dobu se ukáže, že nikam nevede.

Podobně jako pomalé hrany se slepé rychlé hrany po provedení kroku buď zobrazují jako žluté (jejich žlutá je ale tmavší než u pomalých hran) nebo nezobrazují v závislosti na nastavení checkboxu **Skryj žluté.**

Zkuste se vrátit k síti před pročištěním a pak si výsledek operace znovu prohlédnout; je zřejmé, že po pročištění leží *každá* modrá hrana na některé nejkratší nenasycené cestě ze zdroje do spotřebiče a takovou cestu je možno snadno nalézt bez vracení se a navíc *každá* cesta ze zdroje do spotřebiče tvořená modrými hranami je cesta v síti rezerv obsahující minimální počet hran rovný $L(s)$. ◇

Z důvodů, které poznáme později, je následující část algoritmu v pseudokódu na obrazovce souhrnně označena jako nalezení nasyceného toku modrými hranami. Jedná se ale, jak uvidíte při krokování, o cykl sestávající z následujících kroků:

Krok: *Test ukončení cyklu hledání nalezení nasyceného toku modrými hranami*

Během zlepšování toku podél nalezených cest a následného dočišťování - viz následující dva kroky - ubývá modrých hran. Jakmile po zpracování některé cesty zmizí poslední modrá cesta ze zdroje do spotřebiče, pak se všechny modré hrany stanou slepými a následující dočištění je všechny vynechá. Pak budeme muset další cestu hledat mezi hranami, které zatím byly ohodnoceny jako pomalé. Proto se probíraný cykl ukončí a vracíme se znovu na rozdělení

vrcholů do vrstev, které některé hrany dosud označené jako pomalé přehodnotí na rychlé a výpočet obecně může pokračovat. ◇

Krok: *Nalezení a zpracování cesty*

Hledání nejkratší nenasycené cesty bude probráno později, v této scéně se cesta zobrazí okamžitě.

Způsobem, který známe z Ford-Fulkersonova algoritmu nyní nalezneme minimální rezervu na cestě a o tuto hodnotu se sníží rezervy hran na cestě a naopak zvýší rezervy opačných hran. Množina nenasycených hran se tedy změní dvojím způsobem

- pro alespoň jednu hranu nalezené cesty se její rezerva *sníží na nulu* a tedy tato hrana přestane být zobrazována a algoritmem (alespoň dočasně) uvažována; je to hrana nebo hrany s minimální rezervou mezi hranami cesty;

- hranám, které leží v protisměru k hranám cesty, rezerva vzroste; po provedení operace tedy tyto hrany jsou mají kladnou rezervu; některé z nich ji měly kladnou již předtím, ale i pokud byla rezerva nulová, zpracováním cesty vzrostla na kladnou hodnotu a hrana se proto objevila v síti rezerv.

Na úplné provedení tohoto kroku musíme zmáčknout knoflík **Krok** čtyřikrát.

Po prvním klepnutí na knoflík se objeví zeleně vyznačená cesta. Po druhém se určí a zobrazí hodnota Δ minimální rezervy hran cesty. Po třetím klepnutí se barevně zobrazí změny v síti rezerv:

- hrany, kterým rezerva vzrostla, jsou hrany opačně orientované k hranám cesty a zobrazí se ve dvou odstínech červené; hrany, které již předtím byly v síti rezerv, budou světle červené, zatímco hrany, jejíchž rezerva vzrostla z nuly na kladné číslo a které tedy se do sítě rezerv právě dostaly, budou sytě červené;

- hrany cesty, kterým rezerva poklesla, ale nikoli na nulu, takže zůstávají nenasycené, zůstanou zelené a

- hrany cesty, kterým rezerva poklesla na nulu, takže přestanou být nenasycené, budou nyní bílé a neobtažené.

U hran budou také uvedeny změněné rezervy.

Nakonec po čtvrtém klepnutí se barevné označení hran vrátí k obvyklému způsobu: hrany cesty, které přestaly být v síti rezerv, nejsou zobrazeny, ostatní hrany cesty jsou zobrazeny modře.

Opačně orientované hrany, které v minulém kroku byly červené, *jsou pomalé* a proto budou zobrazeny žlutě nebo neznázorněny v závislosti na nastavení checkboxu **Skryj žluté.** Tento fakt je klíčový pro pochopení algoritmu; zpracováním cesty některé hrany ze sítě rezerv mizí, jiné se do ní dostávají, ale nové objeviví se hrany jsou *pomalé* a proto nejsou použitelné pro vytvoření nejkratší cesty ze zdroje do spotřebiče v síti rezerv. Množina modře zbarvených hran se proto zpracováním cesty *zmenší.* ◇

Krok: *Dočištění*

Po zpracování cesty vypadla alespoň jedna modře zobrazená hrana. To může vést k tomu, že některé modré rychlé hrany se stanou slepými - nevede přes ně žádná modrá cesta ze zdroje do spotřebiče. Takové hrany pak v tomto kroku vynecháme stejně tak, jako se to provádělo v kroku čištění uvedeném výše. ◇

Scéna: *Proč krátké cesty*

Projděte si výpočet ještě jednou; krokování je ještě hrubší než v předchozí scéně, protože u zpracování cesty je znázorněn jen výsledek po jejím ukončení. Výpočet si rozdělíme na *fáze*, fáze jsou navzájem odděleny skokem na návěští "loop" v pseudokódu na obrazovce a přepočítání vrstev.

Po celou dobu trvání fáze má nejkratší cesta ze zdroje do spotřebiče v síti rezerv stejnou délku, danou počtem vrstev, označme ji L. Jak jsme viděli už v předchozí scéně, počet cest této délky L každým zpracováním *klesne* alespoň o jednu (vypadne přinejmenším zpracovávaná cesta: alespoň jedna hrana zpracovávané cesty se nasytí) a mohou sice vzniknout nové nenasycené cesty ze zdroje do spotřebiče, ale žádná nová modrá rychlá hrana *nepřibude* a tedy nepřibude ani žádná modrá cesta délky *stejné nebo kratší* než L. V průběhu fáze se postupně všechny původní cesty délky L v síti rezerv přeruší, ale žádná nová cesta délky L nebo kratší nevznikne, takže na začátku nové fáze zjistíme, že vzdálenost ze zdroje do spotřebiče v síti rezerv je *větší* než L.

Zkuste si nyní krokovat výpočet znovu a tvrzení si ověřit. V rohu obrazovky se zobrazuje pořadové číslo fáze a délka nejkratší cesty.

Z uvedeného ihned plyne, že počet fází, neboli počet opakování vnějšího cyklu algoritmu, nemůže být větší než $n - 1$, kde n je počet vrcholů sítě. Prostá cesta ze zdroje do spotřebiče totiž nemůže mít více než $n - 1$ hran, ale alespoň jednu hranu mít musí. Nyní stačí uvážit, že délka nejkratší cesty ze zdroje do spotřebiče může mít jen hodnoty $1, \ldots, n - 1$, ale v každé fázi alespoň o 1 větší.

Kromě toho při každém provedení těla vnitřního cyklu zmizí alespoň jedna modrá hrana, takže počet provedení vnitřního cyklu v rámci jedné fáze, neboli počet cest, které byly v rámci fáze nalezeny a zpracovány, nepřevýší počet

hran, které byly modré při vstupu do cyklu (a tedy je nejvýše roven počtu všech hran sítě).

Scéna: *Proč ne dlouhé cesty*

Tato scéna je shodná s předchozími scénami ukazujícími činnost algoritmu s jedinou, ale podstatnou odchylkou: způsob výběru cesty pro zlepšení toku nepreferuje nejkratší nenasycenou cestu, ale naopak cestu, která se snaží postupovat co nejpomalejším způsobem. Jde vlastně o Ford-Fulkersonův algoritmu, u kterého znázorňujeme rozdělení vrcholů do vrstev.

Na cestě se nyní může objevit i pomalá hrana. V případě výchozí sítě této scény se do cesty dostane hrana vedoucí o dvě vrstvy zpět ke spotřebiči.

Jestliže některá hrana h cesty vede o alespoň dvě vrstvy zpět, hrana h^{op} k ní opačně orientovaná vede o alespoň dvě vrstvy dopředu směrem ke spotřebiči. Znamená to, že h^{op}, která je po zpracování cesty určitě *nenasycená* a tedy patří do sítě rezerv, se stane "superrychlou" nenasycenou hrana, která umožní od zdroje do spotřebiče postupovat rychleji, než tomu bylo dosud možné.

Pokud cesta používá pomalé hrany, vedoucí o jednu vrstvu zpět, nevytvoří to sice kratší nenasycenou cestu, ale může to vést ke vzniku nové a stejně dlouhé nenasycené cesty.

Vidíme, že výběr dlouhých zlepšujících cest by vedl k tomu, že zpracování jedné cesty délky L by mohlo vést k porušení tvrzení, že nevznikají cesty, složené z hran s kladnou rezervou, které by měly délku L nebo menší. Přitom na tomto předpokladu je podstatným způsobem závislý odhad počtu opakování cyklů kódu, který byl uveden v minulé scéně.

Scéna: *Vrstvy*

Tato scéna umožňuje podrobné krokování etapy určování vrstev sítě rezerv. Podrobněji ji ale komentovat nebudeme, protože se jedná o prohledávání sítě do šířky, které bylo podrobně probráno již dříve.

Scéna: *Čištění*

Pročištění sítě je sice rutinní, ale ne zcela jednoduchá operace. Výhodné je provádět ji jak je popsáno dále a jak je možno podrobně krokovat na obrazovce, zatímco ostatní části výpočtu se krokují velice hrubě. Možná si povšimnete, že čištění probíhá způsobem velmi podobným hledání topologického uspořádání acyklického grafu v kapitole 14.

Předně si pro každý vrchol poznamenáváme, kolik modrých rychlých hran z něho vychází a kolik modrých rychlých hran do něho vstupuje. Potom vedeme záznam o vrcholech, ze kterých žádná modrá hrana nevystupuje, ale některé vstupují a záznam o vrcholech s opačnou vlastností (žádná modrá hrana nevstupuje, některé vystupují).

Po rozdělení vrcholů do vrstev a stanovení rychlých modrých hran a žlutých pomalých hran se tyto údaje spočítají zřejmým byť poněkud pomalým způsobem, pak se již jen upravují.

Pak se po sobě zpracují slepé uličky typu "nic nevychází" a typu "nic nevstupuje". Popíšeme zde první operaci, druhá je obdobná.

Do množiny Q vložíme všechny vrcholy, do kterých vstupuje alespoň jedna modrá hrana, ale žádná modrá hrana z něj nevystupuje. Pak dokud je Q neprázdná, opakujeme následující operaci:

z Q se vyjme libovolný vrchol v a pro všechny hrany (u, v) vstupující do v se provede následující:

hrana se přebarví z modré na žlutou, o 1 se sníží údaj o počtu modrých hran vystupujících z u a pokud toto číslo poklesne na 0, vrchol u se zařadí do množiny Q.

Krokujte si výpočet a sledujte, jak se slepé uličky zpracovávají.

Bystrý čtenář si jistě povšiml, že konstrukce a zpracování fronty vrcholů do kterých nevstupují modré hrany je zbytečné: slepé uličky je nutno odstranit, abychom se do nich nedostali při hledání cesty do spotřebiče v čisté síti, vycházejíce přitom ze zdroje, protože jinak by bylo nutno couvat a výpočet by se zpomalil. Vrcholy, do kterých nevstupují modré hrany, jsou ale vrcholy, do nichž se při takovém hledání nedostaneme; proto při hledání cesty nemohou působit komplikace a není tudíž potřeba ztrácet čas jejich odstraňováním.

V appletu je odstraňování slepých uliček typu "nic nevstupuje" zařazeno proto, aby nákres sítě modrých hran byl přehlednější a "čistší". Jelikož zpracování obou front je analogické, nepředstavuje jistě pro čtenáře žádnou zátěž při čtení knihy.

Scéna: *Zpracování cesty*

Tato scéna jen opakuje obdobnou scénu z Ford-Fulkersonova algoritmu. Zařadili jsme ji jen proto, abyste si dobře povšimli, že při hledání cesty v čisté síti se nemůže stát, že bychom zabloudili do slepé uličky, museli se vracet a tím by se hledání prodloužilo. V čisté síti s každým krokem posuneme o jednu vrstvu blíže ke spotřebiči a modré hrany nás do spotřebiče vždy dovedou a proto (na rozdíl od Ford-Fulkersonova algoritmu, ale i od algoritmu Edmondse a Karpa) doba potřebná k nalezení cesty je úměrná její délce.

16.4 Goldbergův algoritmus

Ford-Fulkersonův algoritmus a řada dalších algoritmů, které z něho byly odvozeny, jako je Dinitzův algoritmus, jsou představitelé jednoho typu algoritmů pro toky v sítích, které využívají zlepšující cesty. V této části popíšeme jiný algoritmus, který je základním představitelem algoritmů pracujících s vlnou.

Zatímco metoda zlepšující cesty by se dala snadno vysvětlit bez použití pojmu přebytku toku ve vrcholu a ve Ford-Fulkersonově algoritmu i jeho implementacích, jako je Dinitzův algoritmus, se přebytky objeví jen na velmi krátkou dobu a vždy jen v jediném vrcholu, Goldbergova metoda vlny předpokládá, že po celou dobu výpočtu je v alespoň jednom a většinou v mnoha vnitřních vrcholech kladný přebytek. Budeme ale vyžadovat, aby žádný vrchol (s výjimkou zdroje) neměl nikdy záporný přebytek neboli deficit.

V Goldbergově algoritmu má každý vrchol *výšku*, což je celé nezáporné číslo. Výšku vrcholu v budeme označovat $H(v)$.

Inicializace algoritmu nastaví výšku zdroje na hodnotu rovnou počtu vrcholů sítě, výšky ostatních vrcholů na 0; přebytky všech vrcholů jsou rovny nule, toky hranami jsou nulové a rezervy všech hran jsou rovny jejich kapacitám.

Vytvoření počáteční vlny převede každou hranou vycházející ze zdroje přebytek rovný kapacitě této hrany. Rezervy těchto hran tím poklesnou na nulu.

Algoritmus potom probírá vnitřní vrcholy sítě, které mají kladný přebytek; jestliže se pro takový vrchol v najde z něho vycházející hrana, která má kladnou rezervu a *směřuje dolů* (t.j. její konec má menší výšku než její počátek v), pak hranou převedeme tolik přebytku, kolik je minimum přebytku vrcholu v a rezervy hrany. Nemůžeme totiž převádět více přebytku než ve vrcholu je, protože chceme aby přebytky byly nezáporné, a z kapacitních důvodů nelze převádět více přebytku než je rezerva hrany. Minimum přebytku vrcholu a rezervy hrany je tedy největší množství přebytku, které lze převést.

V okamžiku, kdy již v grafu není žádný vnitřní vrchol s nenulovým přebytkem, výpočet končí.

Algoritmus je mimořádně jednoduchý a zdá se být i přirozený v tom, že přebytek odtéká "z kopce" a jelikož zdroj je výše než spotřebič, tak tok bude téci v požadovaném směru. Bystrý čtenář možná přijde i na to, že pokud se celou počáteční vlnu nepodaří protlačit do spotřebiče, vrcholy ve kterých se zadrhne nakonec vystoupají tak vysoko, že z nich přebytky odtečou zpět do zdroje a v žádném vrcholu nezbude kladný přebytek. Nakonec je jasné, že pokud výpočet skončí, v žádném vnitřním vrcholu sítě nebude nenulový přebytek, takže na konci výpočtu algoritmus vytvoří tok.

Není ovšem patrno, proč by výsledný tok měl mít největší možnou velikost. Jak uvidíme, kdyby zdroj na začátku byl sice vyzvednut, ale do výšky menší než je počet vrcholů, mohlo by se stát, že by výsledný tok skutečně nebyl optimální, což svědčí o tom, že důvody, proč algoritmus dá požadovaný výsledek, nejsou zcela triviální. Není také na první pohled patrno, že by se algoritmus musel vždy zastavit.

Důkaz toho, že algoritmus se vždy zastaví po poměrně malém počtu kroků a že vždy nalezne největší tok v síti, zde neuvádím a čtenáře odkazuji na stan-

dardní učebnici toků v sítích. Mým cílem je pouze ukázat základní myšlenku a mechanismus na základě kterého algoritmus pracuje.

Scéna: *Jednoduchá cesta*

Pro pochopení činnosti Goldbergova algoritmu je velmi užitečné ukázat si jeho výpočet na několika jednoduchých případech. Podobně jako dříve bude přebytek ve vrcholu znázorňován sloupečkem odpovídající výšky.

První případ je síť představovaná prostou orientovanou cestou, jejíž všechny hrany mají stejné kapacity, v tomto případě rovné 1. Vzhledem k linearitě sítě je snadné obrázek nakreslit v 2D i s vystižením výšek vrcholů, které odpovídají výšce na obrazovce.

Procházejte si výpočtem krok za krokem pomocí knoflíku **Krok** a sledujte průběh výpočtu.

V kroku vytvoření počáteční vlny se (jedinou) hranou vycházející ze zdroje přenese do prvního vnitřního vrcholu sítě přebytek rovný kapacitě hrany, v našem případě velikosti 1, zdroj má deficit 1. Hrana se tím nasytí (t.j. její rezerva klesne na 0), opačná hrana získá rezervu 1.

V následujícím kroku má kladný přebytek pouze první vnitřní vrchol, ale hrana z něj jdoucí vlevo strmě stoupá vzhůru a hrana vpravo je vodorovná, proto přebytek nelze převést. Podle algoritmu je tedy vrchol zvednut.

Po zvednutí hrana vedoucí z vrcholu doprava klesá a má dostatečnou rezervu a proto se jí převede přebytek o jeden vrchol doprava. Druhý vrchol musí být nejprve zvednut, pak postoupí přebytek dále doprava a tímto způsobem postupuje, dokud nedoteče do spotřebiče. Vzhledem ke kapacitám hran se pokaždé podaří převést přebytek celý a proto má výsledný tok velikost rovnou 1.

Scéna: *Cesta s překážkou*

Tato síť zjevně připouští pouze tok velikosti 1, protože pravou hranou více toku nepřejde, ale na počátku ze zdroje vytéká levou hranou tok velikosti 2, který vytvoří přebytek velikosti 2 v prostředním vrcholu. Střední vrchol musí být nejprve zvednut, pak z něj může odtéci část přebytku do spotřebiče.

Pravou hranou ale odteče jen polovina přebytku, druhá ve vrcholu zůstane a nemůže odtéci ani vpravo, kde je hrana již natrvalo nasycena, ani vlevo, kde hrana jde prudce do kopce.

Střední vrchol proto stoupá vzhůru. V okamžiku, kdy se dostane o jednu hladinu nad zdroj, pak všechen přebytek odteče vlevo zpět do zdroje, protože hrana směřující vlevo dolů má patřičnou kapacitu.

Tato scéna ukazuje základní mechanismus, jak odstranit tu část počátečního vlny, kterou se nepodaří dopravit do spotřebiče.

Scéna: *Dlouhá cesta s překážkou*

Tato scéna je podobná předchozí, ale síť je tvořena dlouhou cestou, ve které kapacita poslední hrany je menší než jsou kapacity hran předchozích.

Výpočet probíhá podobně; počáteční vlna postupně dorazí až těsně před spotřebič, kde její část projde do cíle, ale další část se odrazí zpět. Způsob jak stoupají vzhůru vrcholy oblasti, ve které se zbytkový tok přelévá z jedné strany na druhou, je nyní podstatně zdlouhavější a složitější než v předchozí scéně, ale zásadní odlišnost zde nevzniká.

Způsob zvedání oblasti bez dočasného odtoku by si zasloužil zlepšení, ale při analýze algoritmu se ukáže, že obecně výpočetnímu času dominuje doba věnovaná nenasyceným převodům přebytku, takže úpravy způsobu zvedání vrcholů, které by mohly algoritmus značně zkomplikovat s nepříliš jistým efektem na celkovou výpočetní dobu, se obvykle neprovádějí.

Scéna: *Cesta s překážkami*

Scéna se podobá předchozí, ale míst zúžení cesty je více a nesousedí přímo se spotřebičem. V tomto případě vznikne více vrcholů s kladným přebytkem a proto applet implementuje několik různých strategií, jak vybrat jeden z nich, protože výše uvedené schéma výběr nespecifikuje.

V závislosti na ovladači bude z vnitřních vrcholů s kladným přebytkem vybrán nejstarší, nejmladší, nejvyšší, nejnižší nebo náhodně zvolený (a v případě výběru nejvyššího a nejnižšího se v případě neurčitosti vybere libovolný z kandidátů s extrémní výškou).

Je vidět, že i když výpočet obecně pro jednotlivé volby probíhá dosti odlišně, žádná z možností nepřináší rozhodující výhody (přinejmenším v tomto příkladě).

Scéna: *Síť s propustnými větvemi*

V této scéně síť není prostá cesta, ale ve vrcholu uprostřed se rozdělí do dvou větví. Je použito 3D zobrazení. Pomocí myši je možno síť natáčet a naklápět (stiskněte knoflík myši a táhněte jí vodorovně nebo svisle).

Tok vypuštěný ze zdroje napřed vstoupí do jedné větve, pak se jeho část odrazí zpět, v rozvětvení vrácený podíl toku vstoupí do druhé větve a je úspěšně dopraven do spotřebiče. Jelikož vrchol rozvětvení byl dosti nízko vzhledem ke zdroji, nemohlo dojít k navrácení toku, který neprošel první větví, zpět do zdroje, ale vše vstoupilo do druhé větve.

V této scéně byl vždy jediný vnitřní vrchol s nenulovým přebytkem a proto není rozdíl mezi variantami výběru vrcholu s nenulovým přebytkem.

Scéna: *Síť s polopropustnými větvemi*

Scéna se podobá předchozí, ale větve nemají ani dohromady dostatek kapacity pro převedení počátečního toku do spotřebiče. Tok, který se nepodařilo odvést do spotřebiče je proto nakonec vrácen do zdroje.

Scéna: *Síť se slabými větvemi*

Scéna se zase podobá předchozí, ale kapacity hran směrem ke spotřebiči klesají, proto se do vrcholů dostává více vlny, než je možno odvést. Celkový průběh výpočtu je v zásadě podobný předchozí scéně, ale postupně obecně vzniká velké množství vrcholů s přebytkem. Různé varianty volby vrcholu s přebytkem proto jsou dosti odlišné. Zkuste si výpočet pro všechny varianty (volba na ovladači) a sledujte rozdíly mezi variantami.

Scéna: *Výška zdroje*

V této scéně si ukážeme, že pokud by zdroj byl umístěn do výšky jen o málo menší než je počet vrcholů, nemusel by Goldbergův algoritmus nalézt optimální řešení (neboli největší tok). Pokud bychom neuvažovali zdroj, je graf představován cestou a po počátečním převodu přebytku všemi hranami vycházejícími ze zdroje bude ve všech vrcholech této cesty kladný přebytek. Zdroje je ale o tři hladiny níže než je počet vrcholů.

Uvidíte, že když přebytky budeme převádět do spotřebiče počínaje "od konce" (což je v souladu s algoritmem), pak když se ke konci výpočtu chceme zbavit přebytku z prvního vrcholu cesty, toho, který je nejdále od spotřebiče, musíme jej zvednout nad následující vrchol, abychom přebytek mohli poslat dál. Tím se ale vrchol dostane se také nad příliš nízko posazený zdroj a přebytek může odtéct zpátky do zdroje a algoritmus se zastaví.

Přitom ale pro přebytek, který jsme zbytečně vrátili do zdroje, stále byla k dispozici volná cesta do spotřebiče, takže algoritmus nenajde největší možný tok.

Povšimněte si, že pokud by zdroj byl jen o dvě hladiny výše, k nežádoucímu předčasnému ukončení výpočtu by nedošlo a výpočet by zdárně nalezl největší tok. Lze dokázat, že pokud je zdroj ve výšce rovné počtu vrcholů, je dostatečně vysoko k tomu, aby algoritmus vždy určil největší tok. Důkaz ale v této knize neuvádím, čtenáře odkazují na standardní učebnici toků v sítích.

Scéna: *Výběr dolního vrcholu*

Jak bylo řečeno, Goldbergův algoritmus nepředepisuje, který vrchol s přebytkem se vybírá. Nejlepší známý odhad časové složitosti v nejhorším případě má varianta, která vždy vybírá nejvyšší vrchol (nebo libovolný z nejvyšších). V této a následující scéně ukážeme, jakou výhodu má volba nejvyššího vrcholu před volbou nejnižšího vrcholu a dalšími variantami algoritmu.

Aby bylo možné ukázat rozdíly variant na co nejjednodušší síti, zvolíme poněkud násilný postup. Ze začátku se vrcholy s přebytkem volí nikoli jako nejvyšší nebo nejnižší, ale tak, aby se vytvořila konfigurace, na které bude rozdíl mezi variantami velmi výrazný. Od okamžiku vytvoření konfigurace se již bude volit buď vždy nejnižší vrchol (v této scéně) nebo vždy nejvyšší vrchol (v následující scéně).

Prvním klepnutím na knoflík **Krok** se neprovede jediný krok, ale série kroků, která vytvoří výše zmíněnou konfiguraci. Pokud se ale na ovladači zvolí **Krátký krok** nebo **Střední krok** namísto **Dlouhý krok,** pak je možno sledovat, jak se do konfigurace dostaneme (v případě krátkého kroku to bude trvat dosti dlouho). Od okamžiku vytvoření konfigurace pak stisknutí knoflíku **Krok** provede jediný přenos přebytku nebo zvednutí vrcholu, bez ohledu na nastavení délky kroku.

Stiskněte tedy knoflík **Krok** a tím vytvoříte výše zmíněnou konfiguraci, kde každý vnitřní vrchol má kladný přebytek velikosti 1 a tyto vrcholy vytvářejí cestu končící ve spotřebiči s klesající výškou vrcholů. Varianta volící nejnižší vrchol převádí přebytek každého vrcholu do spotřebiče *zvlášť* cestou, která může být i velmi dlouhá. Myslím, že celý výpočet nedokončíte, protože trvá hodně dlouho, ale jistě pochopíte proč je tak pomalý podstatně dříve, než bude ukončen.

Scéna: *Výběr horního vrcholu*

Zde opakujeme výpočet předchozí scény s tím, že z uvedené předělové konfigurace postupujeme tak, že vybíráme vždy nejvyšší vrchol s přebytkem.

První krok po dosažení konfigurace přenese přebytek nejvyššího vnitřního vrcholu do vrcholu za ním následujícího. Převedený přebytek se přičtením *sloučí* s přebytkem vrcholu, do kterého se dostal a v následujícím kroku se sloučené přebytky převádí dále společně. Totéž se stane i v dalších vrcholech. Vidíte, že tato varianta přebytky nezpracovává odděleně, ale slévá je dohromady a přenáší aglomerovaný přebytek v jedné operaci přenosu, čímž se dosáhne výrazné úspory výpočetního času.

Scéna: *Goldbergův algoritmus*

Závěrečná scéna umožní rekapitulaci všeho, co bylo vyloženo. Je možné volit různé příklady nebo si vytvořit svoji síť, vybrat si variantu algoritmu i délku kroku a sledovat výpočet. Pohyby myší také umožňují síť naklápět a rotovat. V této knize neprovádíme podrobnou analýzu správnosti nebo časové složitosti algoritmu a čtenář je odkazován na standardní učebnice grafových algoritmů.

Kapitola 17

Minimální řez a vlastní vektory

Tento applet je hodně odlišný od všech ostatních appletů v knize. Předně algoritmus, který zde uvedu, nezaručuje nalezení optimálního řešení problému. V této knize nerozebírám podrobněji teorii výpočetní složitosti, ale pokud o ní něco znáte, jistě víte, že úloha nalezení minimálního řezu v grafu je NP-úplná, což zhruba znamená, že není znám žádný dostatečně rychlý algoritmus pro hledání optimálního řezu v grafu a má se za to, že ani neexistuje.

Kromě toho v případě tohoto algoritmu nebudeme podrobně rozebírat teorii, na které je založen. Jedná se o spektrální teorii grafů a například důkaz vět o vztahu hodnoty rozdílu prvního a druhého vlastního čísla matice sousednosti grafu se stupněm souvislosti grafu zdaleka přesahuje ambice a možnosti této knihy.

Nakonec procházka tímto appletem má jedinou scénu a spočívá v pouhém prohlížení různých vývojových stádií grafu lišících se velikostí minimálního řezu a sledování odpovídajícího vlastního vektoru odpovídajícího druhému vlastnímu číslu.

Applet byl přesto zařazen do knihy nejen proto, že hledání minimálního řezu patří mezi z praktického hlediska nejdůležitější problémy teorie grafů a spektrální heuristiky jsou považovány za nejlepší známé způsoby jak hledání provádět, ale především proto, že vlastnosti vlastních čísel a vektorů matic obecně a matic sousednosti grafu speciálně jsou považovány za teorii, která je sice matematicky obtížná a přitom elegantní, ba dokonce krásná, ale pro praxi zcela nepotřebná. Zatímco výrok o eleganci a kráse je naprosto výstižný, předsudek o nepraktičnosti a neužitečnosti spektrálních teorií je způsoben především neznalostí a hlavním cílem této kapitoly je tuto neznalost alespoň částečně rozptýlit. Doufám také, že uvidíte, že myšlenky, na kterých je spektrální

teorie založena jsou překvapivě jednoduché a snadno pochopitelné.

Z důvodu přístupnosti výkladu se omezím na regulární neorientované grafy, ale většina výsledků se dá zobecnit i na obecnější třídy grafů (ale za cenu podstatně větší složitosti teorie). Graf je regulární, pokud každý vrchol má stejný stupeň, tedy je spojen se stejným počtem jiných vrcholů grafu. V této kapitole budeme stupeň grafu označovat symbolem D.

Během procházky appletem budeme pouze sledovat některá fakta a vlastnosti grafu, aniž bych předkládal jejich vysvětlení. Nebudu ani ukazovat, jak se vlastní čísla a vlastní vektory spočítají, takže čtenář, který způsoby jejich výpočtu nezná, ať si prostě představuje, že si je může snadno spočítat v libovolném standardním matematickém systému jako je (abecedně) Mapple, Mathematica nebo Matlab a nebo určit některým z jednoduchých či složitějších programů, které najde v literatuře nebo na Internetu. Doufám ale, že shledáte vztah vlastností grafu a jeho spektra natolik zajímavý a užitečný, že vás podnítí k podrobnějšímu studiu této pozoruhodné teorie.

Scéna: *Minimální řez regulárního grafu*

Úloha, která má být řešena, je jednoduchá: je dán neorientovaný a v našem případě regulární graf G se sudým počtem vrcholů. *Řez* grafu pro nás bude rozdělení množiny vrcholů grafu na dvě stejně velké části. *Velikost řezu* je počet hran, které mají své konce v opačných částech rozdělení, tedy které byly řezem "přeseknuty". Je třeba nalézt řez grafu, který má nejmenší možnou velikost. Jak bylo řečeno výše, jedná se o obtížnou úlohu, kterou pro větší grafy umíme řešit jen přibližně.

Na obrazovce je nakreslen graf stupně 4 se 64 vrcholy. Je možno vygenerovat nový graf s případně změněnými parametry, které se zadávají na ovladači. Graf je nakreslen náhodně, ale tak, aby měl řez absolutně nejmenší možnou velikost - nulovou: je jasně vidět, že mezi levou a pravou polovinou vrcholů nevede žádná hrana.

U pravého okraje okna jsou uvedena první dvě vlastní čísla matice sousednosti grafu: největší vlastní číslo λ_1 nahoře a druhé největší vlastní číslo λ_2 pod ním. Později si ukážeme, že největší vlastní číslo λ_1 neorientovaného regulárního grafu stupně D je rovno D a že u absolutně separovaného grafu, jaký je nakreslen na obrázku, je toto vlastní číslo (alespoň) dvojnásobné, takže za druhé vlastní číslo λ_2 je považováno také D, v tomto případě 4.

Pod grafem je graficky znázorněn vlastní vektor odpovídající druhému vlastnímu číslu. Pokud jsou první a druhé vlastní číslo stejné, bývá za druhý vlastní vektor považován ten z vlastních vektorů odpovídajících D, pro který je součet složek roven 0 (více později). Kladné složky jsou kresleny zeleně, záporné složky černě.

Teoreticky by se mohlo stát, že vlastní číslo D by mělo větší násobnost než 2 a vlastních vektorů se součtem složek 0 by mohlo být více. Při našem způsobu generování grafu je to ale mimořádně nepravděpodobné.

Složky vektoru jsou přirozeným způsobem asociovány s vrcholy grafu. To je v našem případě vystiženo tím, že složka odpovídající vrcholu je nakreslena přímo pod ním. (Vrcholy jsou posunuty do takových poloh, aby se složky nepřekrývaly a aby jejich rozestupy byly konstantní).

Je okamžitě vidět, že nalezení minimálního (nulového) řezu lze provést tak, že se určí druhý vlastní vektor matice grafu a složky se rozdělí podle znaménka odpovídající složky vlastního vektoru. Pro nesouvislý graf se dvěma stejně velkými komponentami by určování komponent pomocí druhého vlastního vektoru bylo směšné, protože stejný výsledek se dá určit daleko jednodušími metodami (například prohledáváním). V dalších krocích se ale spektrální metoda brzo stane zcela přiměřená složitosti problému.

Klepněte nyní na knoflík **Krok.** Graf se změní následujícím způsobem: náhodně se vybere hrana $u - v$ v levé polovině a hrana $w - z$ v pravé polovině vrcholů, tyto hrany se odeberou a přidají se hrany $u - w$ a $v - z$. Touto přesmyčkou vznikne graf, který je již souvislý, tedy v něm není řez velikosti 0, ale na obrazovce dané rozdělení na levou a pravou polovinu vrcholů dává řez velikosti 2, tedy přes hranici řezu přecházejí dvě hrany. (Regulární graf sudého stupně nemůže mít řez liché velikosti, například 1).

Konstrukce grafu i přesmyčky hran jsou prováděny náhodně. Proto nemohu některé jevy předpovědět se stoprocentní jistotou. Pravděpodobnost, že ale bude vše probíhat tak, jak zde předpovím, je tak blízká 1, že předpovědi klidně podáván a jsem si zcela jist, že se ukáží jako pravdivé. Tuto lekci přednáším řadu let, nikdy se nestalo, že by něco z toho, co zde tvrdím, nebyla pro znázorněný graf pravda (a dá se dokázat, že bych tak mohl ve většině případů činit ještě několik století a přesto oprávněně doufat, že se nikdy nespletu).

Podívejme se nyní, jak se přesmyčka projeví na vlastních číslech a vektorech. Největší vlastní číslo bude po celou dobu rovno stupni vrcholů. Druhé největší vlastní číslo se ale změnilo a už je o něco menší než největší vlastní číslo.

Složky vlastního vektoru odpovídajícího druhému vlastnímu číslu nadále mají stejné znaménko na levé polovině vrcholů a opačné znaménko na pravé polovině vrcholů. Optimální řešení lze tedy i nyní nalézt tak, že se vrcholy rozdělí podle znaménka odpovídající složky druhého vlastního vektoru.

I u vlastního vektoru ale nastala jistá změna. Zcela určitě naleznete čtyři složky vlastního vektoru, které jsou v absolutní hodnotě zřetelně menší než ostatní složky. Dvě by měly být v levé polovině vektoru, další dvě v pravé polovině. Klepněte na některou ze složek. Zvýrazní se (zbarví do červena) nejen složka, ale i odpovídající vrchol a hrany z něj vycházející.

Nyní určitě zjistíte, že vrcholy odpovídající složkám druhého vlastního vektoru, které mají nápadně malou absolutní hodnotu (dvě zelené kladné a dvě černé záporné) jsou přesně ty čtyři vrcholy, které mají souseda v druhé

polovině grafu.

Vlastní vektor mívá i další složky, které jsou v absolutní velikosti menší ve srovnání s většinou složek. Klepnutím na takovou složku určitě zjistíte, že odpovídající vektor sice má všechny své sousedy ve své polovině, ale alespoň z jednoho z jeho sousedů vede hrana do druhé části, je to tedy jeden ze čtyř "defektních" vrcholů. Pokud by takový vrchol měl dva "defektní" sousedy, bude snížení v jeho složce ve vlastním vektoru výraznější, ale zase menší než u defektní čtveřice. Další jemnější rozdíly velikostí složek jednotlivých vrcholů budou komplexním a těžko rozpoznatelným způsobem vystihovat méně významné závislosti, jako je vzdálenost od defektních vrcholů a podobně.

Knoflíkem **Krok** pokračujte dále. Opakovaně se provádí výše popsaná přesmyčka hran, která zvyšuje počet hran, které jsou mezi levou a pravou polovinou. Současně s tím se zvětšuje rozdíl mezi stále stejným největším vlastním číslem a klesajícím druhým vlastním číslem.

Mění se i vzhled vlastního vektoru; rozdělení na kladné a záporné složky indukuje ještě po dosti dlouhou dobu naše pravo-levé dělení, ale hodnoty složek začínají být rozrůzněné. Obecnou tendencí je, že vrchol s malou absolutní hodnotou složky mívá více sousedů na druhé straně grafu, ale hodnotu složky také citelně ovlivňuje "kvalita" těchto sousedů.

Po jisté době se objeví vrchol, jehož složka má opačné znaménko, než by odpovídalo jeho příslušnosti nalevo či napravo. Napravo se mezi zápornými hodnotami může objevit složka, která je malé kladné číslo a nebo se nalevo mezi kladnými složkami objeví jedna, která je záporná, byť má asi malou absolutní hodnotu. Studenti mne obvykle upozorňují na chybu. Nejčastěji ale jde o vrchol, který má dva (nebo obecněji $D/2$) sousedů ve své polovině a stejně sousedů v druhé. U něj pak je zcela lhostejné, zda je vlevo nebo vpravo, velikosti řezu přispívá stejně. Pokud je podobný vrchol i na druhé straně, jejich záměnou by se velikost řezu nezměnila. Pak ale znaménko složky takového vrcholu vlastně může být jakékoli, takže není chybou, jestliže vyjde opačné než by zdánlivě mělo být.

I u takto malého grafu je obtížné nalézt optimální řez a je proto těžké porovnávat jak kvalitní je pravolevé dělení a jaké je hodnota řezu získaného rozdělením vrcholů podle složek druhého vlastního vektoru - buď podle znamének nebo později raději obecněji na horní polovinu a dolní polovinu. Mnohé indicie ale napovídají, že když je spektrální řešení odlišné od pravo-levého, je to nejspíše tím, že nakreslené rozdělení na levou a pravou polovinu již není optimální řez a řešení podané na základě vlastního vektoru většinou bývá lepší.

Právě popsaný spektrální algoritmus se zdá být jako magie; aniž bychom graf podrobněji zkoumali, vložíme jeho matici sousednosti do Matlabu a vzápětí dostaneme mimořádně kvalitní řešení. Zdá se, že je třeba vysokého matematického vzdělání, aby bylo možno pochopit příčiny. Základní myšlenka je

přitom ale takřka triviální.

Představme si, že každému vrcholu v přiřadíme reálné číslo $\theta(v)$ v rozmezí $[-1, 1]$, které budeme nazývat *skóre* vrcholu a která vyjadřuje, jak moc by vrchol měl být nalevo či napravo. Hodnota +1 znamená určitě nalevo, -1 znamená určitě napravo, hodnota kolem nuly znamená, že je skoro jedno, kde leží. Aby bylo co nejméně hran, křižujících řez, je vhodné dát vrcholu vysoké kladné skóre, pokud skóre jeho sousedů jsou také hodně kladné, podobně pro záporné hodnoty. Jinými slovy, pokud sousedi vrcholu leží většinou nalevo (nebo napravo), měli bychom dát vrchol také nalevo (nebo napravo). Těžko si představit něco lepšího, než chtít, aby skóre $\theta(v)$ vrcholu v bylo součtem skóre jeho sousedů, popřípadě vynásobeným jistým konstantním normovacím koeficientem stejným pro všechny vrcholy, tedy

$$\theta(v) = \frac{1}{\lambda} \sum_{w} \theta(w)$$

kde součet je přes všechny sousedy w vrcholu v. Jediná potíž je, že neznámou hodnotu θ pro vrchol v se snažíme určit na základě neznámých hodnot θ pro jeho sousedy. Dostáváme tedy soustavu rovnic (pro všechny vrcholy), a jak již možná vidíte a nebo se dozvíte za chvíli, není to nic jiného, než zápis říkající, že θ je vlastní vektor matice sousednosti grafu G odpovídající vlastnímu číslu λ. Formulace problému jako určování vlastních čísel je tedy zcela přímočará a přirozená a jediné, co není jednoduché, je tuto rovnici vyřešit.

Zbývá jen odpovědět, proč nás zajímá druhý vlastní vektor a nikoliv první. Výše popsaný problém má totiž jedno triviální řešení. Všechny vrcholy dáme do jedné skupiny, například levé a to s naprostou jistotou, tedy každému vrcholu dáme skóre +1. Pak je skóre pro každý vrchol aritmetickým průměrem skóre jeho sousedů (protože zde cokoliv je aritmetickým průměrem čehokoliv). To ale je přesně řešení odpovídající (největšímu) vlastnímu číslu D se souvisejícím vlastním vektorem rovným $(1, 1, \ldots, 1)$, takže nejvyšší vlastní číslo a jeho vlastní vektor popisují triviální řešení, které nás nezajímá. Druhý vlastní vektor je kolmý na první, což znamená, že součet "pravosti" a "levosti" přes všechny vrcholy je nulový, což je blízké tomu, že vrcholy budou podle znaménka rozděleny do dvou zhruba stejně velkých částí.

Jedna poznámka nakonec: není-li graf regulární, tedy různé vrcholy mají různé počty sousedů, pak vlastní vektor, odpovídající největšímu vlastnímu číslu, není konstantní, ale složka vlastního vektoru odráží jak počet sousedů odpovídajícího vrcholu, tak i jejich důležitost (danou zase počtem a důležitostí jejich sousedů). Vrchol je tedy důležitý, když má hodně sousedů a ti jsou také důležití. Na tomto principu je založen PageRank v Google, který byl základem jeho obrovského úspěchu. Je to ale větev spektrální teorie grafů, kterou v této knize z důvodu jejího rozsahu nechci rozebírat.

Nyní ještě dokážeme některá tvrzení, která byla bez důkazu uvedena výše.

Závěr

Nechť G je graf s vrcholy označenými jako $u_1, \ldots, u_n$. Matice $A = (A_{ij})$ se nazývá matice sousednosti grafu G, jestliže to je čtvercová matice řádu $n \times n$, kde n je počet vrcholů grafu, taková že $A_{ij} = 1$ pokud u_i a u_j jsou spojeny hranou, jinak je $A_{ij} = 0$. Pokud je graf neorientovaný, je matice symetrická, tedy $A_{ij} = A_{ji}$ pro každé i a j.

V této kapitole jsme se omezili na regulární grafy stupně D; u nich v každém řádku i každém sloupci jeho matice sousednosti se nachází přesně D jednotek a ostatní prvky jsou rovny nule.

Jestliže $A = (a_{ij})$ je matice řádu $n \times n$ s obecně komplexními koeficienty (například matice sousednosti grafu o n vrcholech) a pro nenulový vektor $v = (v_1, \ldots, v_n)$, který je prvek vektorového prostoru dimenze n nad komplexními čísly a pro komplexní číslo λ platí rovnice

$$Av = \lambda v,$$

neboli

$$\sum_{j=1}^{n} A_{ij} v_j = \lambda v_i \quad \text{pro každé } i = 1, \ldots, n,$$

pak řekneme, že λ je *vlastní číslo* matice A a v je vlastní vektor matice A příslušný vlastnímu číslu λ.

Snadno si ověříte, že libovolný nenulový násobek vlastního vektoru příslušného k λ je také vlastním vektorem příslučným k λ. Jednoduché je také zjistit, že D je vlastní číslo ke kterému přísluší vlastní vektor $(1, 1, \ldots, 1)$.

Důkaz toho, že D je největší vlastní číslo, je sice jednoduchý, ale jde již mimo rámec knihy a neuvádím ho.

Předpokládejme nyní, že množina V vrcholů neorientovaného regulárního grafu stupně D má řez velikosti 0 (tedy nejmenší možná hodnota). To znamená, že V se dá rozdělit na dvě stejně velké disjunktní množiny V_1 a V_2 takové, že neexistuje žádná hrana, která by měla jeden konec ve V_1 a druhý ve V_2. V takovém případě je n sudé a tedy $n = 2m$ pro nějaké přirozené číslo m. Vrcholy množiny V můžeme uspořádat jako $u_1, \ldots, u_n$ tak, že $V_1 = \{u_1, \ldots, u_m\}$ a $V_2 = \{u_{m+1}, \ldots, u_n\}$. V takovém případě ukážeme, že D je alespoň dvojnásobné vlastní číslo a další vlastní vektor, který k němu přísluší a je kolmý na vektor $(1, \ldots, 1)$, je vektor $(1, \ldots, 1, -1, \ldots, -1)$, který je tvořen nejprve m jednotkami a pak m čísly -1. Matici sousednosti má v tomto případě totiž tvar

$$\begin{pmatrix} A_1 & 0 \\ 0 & A_2 \end{pmatrix}$$

kde symbol 0 označuje matice řádu $m \times m$ s nulovými koeficienty, neboť pokud $i \leq m < j$ nebo $i > m \geq j$, pak $A_{ij} = 0$. Podgrafy grafu G určené

množinami V_1 a V_2 jsou oba zase regulární grafy stupně D a jejich matice sousednosti jsou A_1 a A_2, takže D je vlastní číslo obou matic a $v = (1, \ldots, 1)$, ale pochopitelně také $w = (-1, \ldots, -1) = -v$ jsou jejich vlastní vektory, odpovídající vlastnímu číslu D. Vektor $(1, \ldots, 1, -1, \ldots, -1)$ můžeme psát v řádkovém tvaru jako (v, w), podobně i ve sloupcovém tvaru a dostáváme

$$\begin{pmatrix} A_1 & 0 \\ 0 & A_2 \end{pmatrix} \begin{pmatrix} v \\ w \end{pmatrix} = \begin{pmatrix} Dv \\ Dw \end{pmatrix} = D \begin{pmatrix} v \\ w \end{pmatrix}$$

Pro ty, kteří mají rádi podrobné výpočty důkaz zopakujeme ještě jednou s tím, že matice $A = (A_{ij})$ splňuje $A_{ij} = 0$ pokud $i \leq m < j$ nebo $i > m \geq j$ a vektor $z = (z_1, \ldots, z_n)$ je definován takto: $z_i = 1$ pro $i = 1, \ldots, m$ a $z_i = -1$ pro $i = m + 1, \ldots, n$:

$$\sum_{i=1}^{n} A_{ij} z_j = \sum_{i=1}^{m} A_{ij} z_j = \sum_{i=1}^{m} A_{ij} = \sum_{i=1}^{n} A_{ij} = D = D z_i \quad \text{pro } i \leq m,$$

$$\sum_{i=1}^{n} A_{ij} z_j = \sum_{i=m+1}^{n} A_{ij} z_j =$$

$$= \sum_{i=m+1}^{n} -A_{ij} = -\sum_{i=1}^{n} A_{ij} = -D = D z_i \quad \text{pro } i > m,$$

což dohromady znamená, že $Av = Dv$.

Jestliže v grafu existuje řez velikosti 0, pak druhé největší vlastní číslo je rovno D a je tedy stejně velké jako největší vlastní číslo. Navíc nalezení řezu je možno provést tak, že nalezneme vlastní vektor odpovídající číslu D a je kolmý na základní vlastní vektor $(1, \ldots, 1)$ a vrcholy rozdělíme podle znaménka odpovídající složky tohoto vlastního vektoru.

Z klasické spektrální teorie matic naopak vyplývá, že jestliže je rozdíl mezi prvním a druhým vlastním číslem velký, je graf hodně souvislý (ve smyslu, který zde nebudeme popisovat), takže i nejmenší řez v grafu bude velký. Tato teorie však přesahuje zaměření naší knihy; říká ale, že čím je větší velikost minimálního řezu (neboli čím je graf "souvislejší"), tím je "díra" mezi prvním a druhým vlastním číslem větší, což je přesně to, co jsme viděli během procházky appletem.

Část IV

Aritmetické algoritmy

V této části knihy probereme dva problémy, které spolu příliš nesouvisí, ale mají dvě společné vlastnosti: jednak je možno oba považovat za aritmetické algoritmy a kromě toho jsou oba z praktického hlediska velmi významné.

První úlohou je sčítání čísel zadaných v dvojkové soustavě. Je to základní aritmetická operace, kterou musí počítačové procesory provádět co možná nejrychleji. Snažíme se přitom počítat paralelně, tedy provádět co nejvíce operací současně. Klasický algoritmus pro sčítání, který znáte ze základní školy, ale je pro paralelní počítání zcela nevhodný a proto zde ukážeme jiný postup, který je výrazně rychlejší, protože umožní velmi rozsáhlé použití paralelismu.

Druhou úlohou je počítání přímé a inverzní diskrétní Fourierovy transformace (FT). Jedná se o klasickou úlohu; spojitá verze Fourierovy transformace je v matematice využívána už velmi dlouho (Fourier, po kterém je pojmenována, žil v letech 1768-1930). Fourierova transformace je především důležitá pro spektrální analýzu signálů a speciálně diskrétní varianta FT je používána pro spektrální analýzu digitalizovaných akustických i jiných signálů. Má ale řadu dalších použití, například na kosinovou transformaci, která je užívána pro JPEG kompresi obrazového signálu, je možno se dívat jako na úpravu Fourierovy transformace tak, aby používala jen reálná a nikoliv obecná komplexní čísla.

V poslední kapitole této části bude probrán klasický algoritmus nazývaný rychlá Fourierova transformace (FFT, z angl. Fast Fourier Transform), což je metoda, díky jejíž výpočetní rychlosti a jednoduchosti je využívání diskrétní Fourierovy transformace tak rozšířené. Výhodou algoritmu je, že jej lze používat jak pro přímou, tak i inverzní FT.

Jelikož aplikace Fourierovy transformace nejsou obecně dostatečně známy, je před kapitolu o FFT zařazena ještě jedna kapitola, která pojednává o použití diskrétní FT pro spektrální analýzu periodického digitalizovaného signálu.

Kapitola 18

Sčítání čísel

Scéna: *Operandy*

Naším úkolem je sečíst dvě čísla napsaná v binární soustavě. V první scéně je ukázáno, jak se v appletu pracuje se sčítanci.

Oba sčítance mají 32 číslic; tento počet lze v rozumných mezích měnit v poli **N=** na jinou mocninu 2. Knoflíky **RND** lze 1. nebo 2. sčítanci přiřadit jinou náhodnou hodnotu. V řadě případů také bude užitečné nastavit jeden sčítanec jako komplement druhého (0 proti 1 a 1 proti 0); to se provede odpovídajícími knoflíky **KMPL.** Nakonec lze hodnotu každé číslice změnit (z 1 na 0 a z 0 na 1) klepnutím myší.

Scéna: *Algoritmus ze základní školy*

Zdálo by se, že není třeba o sčítání podrobněji mluvit. Algoritmus se žáčci učí již na základní škole a zná jej každý gramotný člověk a způsob sám je velmi jednoduchý a na první pohled i rychlý.

Tato scéna umožňuje krokovat výpočet podle základní školy a ukazuje, že postup je ve skutečnosti pro větší čísla poměrně zdlouhavý (zkuste si jej pro 64 bitů). Jistě není třeba dodávat, že červená šipka označuje přenos do vyššího řádu a zelená tečka naopak naznačuje, že k přenosu nedošlo.

Při sčítání dvou čísel algoritmus postupuje zprava doleva a každý krok je vázán na výsledek předchozího (konkrétně na to, zda dojde k přenosu do vyššího řádu). Počet kroků je proto roven počtu číslic sčítanců.

Scéna: *Paralelní algoritmus ze základní školy*

V moderních procesorech je k dispozici dostatečné množství logických obvodů, takže (na rozdíl od lidského počtáře) mohou provádět velké množství

operací současně a tím urychlovat výpočet. Při současné šířce slova v počítačích, kdy 64 bitů se stává standardem, je takové zrychlení opravdu žádoucí. Zdá se ale, že elementární algoritmus pro sčítání nelze paralelním prováděním operací nijak urychlit.

Je zřejmé, že nejobtížnější část výpočtu je určit, ve kterých sloupcích dochází k přenosu. Pokud to víme, dá se výsledný součet určit *jediným* paralelním krokem, kdy pro každý sloupec sečteme dvě číslice sloupce a přenos z nižšího řádu modulo 2, neboli výsledek ve sloupci je 1 právě když buď 1 nebo 3 z těchto hodnot jsou rovny 1.

Při podrobnějším pohledu na algoritmus pro sčítání je zřejmé, že pro některé sloupce umíme rozhodnout, zda z nich bude vycházet přenos do vyššího řádu, aniž bychom čekali na výsledek z řádů nižších.

- Jestliže ve sloupci oba sčítance mají hodnotu 1, pak z takového sloupce bude přenos vycházet *vždy*, bez ohledu na to, zda z nižších řádů přenos přijde nebo ne. Budeme říkat, že sloupec *generuje*.

- Jestliže ve sloupci oba sčítance mají hodnotu 0, pak z takového sloupce přenos *nikdy* nevyjde, i kdyby do něho vstoupil přenos z nižšího řádu. Budeme říkat, že sloupec *absorbuje*.

- Nakonec jestliže sloupec je komplementární, tj. v jednom sčítanci je 0 a v druhém 1, pak sloupec vytváří přenos pouze pokud vytváří přenos sloupec napravo od něj. Budeme říkat, že sloupec *přenáší*.

Povšimněte si také, že sloupec zcela vpravo nemá co přenášet, protože napravo od něj již další sloupec není. U pravého sloupce je proto vždy možno rozhodnout o přenosu bez znalosti hodnot v jiných sloupcích.

Pro rozdělení sloupců do typů není třeba znát informaci v jiných sloupcích a proto je snadno provedeme pro všechny sloupce současně a nezávisle v jednom paralelním kroku.

Zkuste nyní algoritmus krokovat. V prvním kroku se u sloupce vpravo a u všech sloupců, které generují nebo absorbují, určí, zda z nich vychází přenos. Zbývající (přenášející) sloupce vytvoří "ostrůvky" nerozhodnutých sloupců, oddělené rozhodnutými sloupci.

Šířka těchto ostrůvků bývá výrazně menší, než je šířka celých čísel a dá se dokázat, předpokládajíce náhodné a navzájem nezávislé hodnoty číslic sčítanců, že generujících a absorbujících sloupců bývá okolo jedné poloviny všech sloupců (jednoduchý důkaz) a šířka největšího ostrůvku je obvykle řádově kolem logaritmu šířky slova (značně složitý důkaz).

Můžete si zkusit opakovaně generovat náhodné sčítance a provádět první krok algoritmu a sledovat, jak ostrůvky nerozhodnutých sloupců vypadají.

Nyní budeme pokračovat krokování. V druhém kroku je možno o přenosu rozhodnout v pravých sloupcích ostrůvků a v každém dalším kroku se ostrůvky

postupně zmenšují zprava doleva, o 1 sloupec v každém kroku. Současně se přitom zpracovává tolik sloupců, kolik je dosud nerozhodnutých ostrůvků a doba paralelně prováděného výpočtu je proto dána maximální šířkou ostrůvku a je proto obvykle výrazně menší, než u základní varianty sčítacího algoritmu.

Scéna: *Paralelní algoritmus - nejhorší případ*

Jak bylo řečeno v předchozí scéně, doba nutná k paralelnímu provedení sčítání podle vykládaného algoritmu je dána šířkou největšího ostrůvku a je tedy silně závislá na konkrétním tvaru sčítanců.

V nejlepším případě, pokud například sčítáme dvě stejná čísla, ostrůvky vůbec nevzniknou a rozhodnutí o všech přenosech se dostane v jednom kroku. I v průměrném případě je algoritmus velmi rychlý.

V nejhorším případě však bohužel je tento algoritmus stejně rychlý (či lépe řečeno pomalý) jako sériově prováděný algoritmus ze základní školy. Nastává to v případě, kdy všechny sloupce (s případnou výjimkou sloupce vpravo) jsou komplementární. Pak totiž vznikne jen jeden ostrůvek, který má šířku o 1 menší než je šířka sčítanců a proto výpočet probíhá úplně stejně jako v základním algoritmu a paralelismus je zcela nevyužit.

Tento případ generuje tato scéna; zkuste si výpočet krokovat, ale myslím, že jej ani nedokončíte, protože trvá příliš dlouho a nic nového se při něm nedozvíte.

Scéna: *Bloky*

Nyní začneme vysvětlovat jiný algoritmus, obvykle označovaný jako carry look-ahead, který zaručeně i v nejhorším případě počítá velmi rychle, rychlostí srovnatelnou s rychlostí, kterou paralelizovaný elementární algoritmus počítá v obvyklém případě, a tedy podstatně rychleji než elementární algoritmus v základním provedení.

Základním prvkem pro nový algoritmus je blok. *Blok* je souvislý soubor sloupců, na obrazovce je označen žlutým obdélníkem. Šířku bloku je možno na obrazovce měnit tažením jeho levé nebo pravé strany myší. Polohu bloku lze měnit tažením bloku myší za libovolné jiné místo než je jeho pravý a levý okraj nebo okénko s číslicí. Je též možno požádat o vytvoření nového (náhodného) bloku.

U bloku nás pro dané hodnoty sčítanců bude zajímat, zda z jeho nejvyššího (tj. nejlevějšího) sloupce vychází přenos. Zda k tomu dojde však se budeme snažit určit pouze na základě těch číslic sčítanců, které jsou uvnitř bloku. To pochopitelně vždy nepůjde a proto budeme, podobně jako už jsme to učinili výše, rozlišovat následující tři případy:

- blok *generuje*: některý sloupec bloku obsahuje dvě číslice 1 a všechny sloupce bloku, které leží nalevo od něho, jsou komplementární (obsahují jednu 1 a jednu 0); tento případ budeme označovat symbolem '+'

- blok *absorbuje*: některý sloupec bloku obsahuje dvě číslice 0 a všechny sloupce bloku, které leží nalevo od něho, jsou komplementární (obsahují jednu 1 a jednu 0); tento případ budeme označovat symbolem '-'

- blok *přenáší*: všechny sloupce bloku jsou komplementární; tento případ budeme označovat symbolem '<' (s výjimkou případu, kdy blok zahrnuje sloupec zcela vpravo, kdy budeme také používat symbol '-').

Je jasné, že tyto tři případy se navzájem vylučují a jeden z nich vždy nastává. V prvním případě, kdy blok generuje, ze sloupce obsahujícího dvě 1 vyjde přenos a je přenášen komplementárními sloupci až k levému okraji bloku a ven z něho, proto z levého sloupce bloku přenos vychází. V druhém případě sloupec se dvěma nulami absorbuje případný přenos, který by do tohoto sloupce vstoupil z nižšího řádu a sloupce nalevo od něj samy o sobě přenos vytvořit nejsou schopny.

V nejzajímavějším třetím případě sloupce bloku samy přenos nevytvoří, ale pokud ze sloupce stojícího vpravo od bloku do bloku přenos vstoupí, sloupce bloku přenos přenesou celým blokem, až nakonec vystoupí z levého sloupce bloku dále. Jestliže však vpravo od bloku již žádný jiný sloupec neleží, je zaručeno, že nebude co přenášet, což vysvětluje výjimku v označování přenášejícího bloku.

Vyzkoušejte různé hodnoty sčítanců a šířku i polohu bloku a sledujte, která z možností '+', '-' a '<' nastává a snažte si ji zdůvodnit.

Scéna: *Jednosloupcový blok*

Jak již bylo naznačeno výše, výpočet hodnoty bloku tvořeného jedním sloupcem je jednoduchý: sloupec obsahující dvě 1 generuje, pokud obsahuje dvě 0, pak absorbuje a komplementární blok zcela vpravo negeneruje a jinak přenáší.

Ověřte si tyto možnosti pro různé hodnoty sčítanců a polohu bloku, který je možno myší táhnout jako tomu bylo v předchozí scéně (ale nelze měnit jeho šířku).

Scéna: *Široký blok*

Hodnotu bloku složeného z více než jednoho sloupce lze určit snadno podle definice: Postupujíce zleva doprava, najdeme první nekomplementární sloupec bloku. Obsahuje-li dvě 1, blok generuje, jinak absorbuje. Pokud je ale blok složen jen z komplementárních sloupců, pak přenáší nebo negeneruje podle toho, zda zasahuje až do pravého sloupce sčítanců.

Tento postup je sice jednoduchý, ale sekvenční a v nejhorším případě (komplementární operandy) by vedl ke stejně pomalému výpočtu jako předchozí dvě metody.

Budeme proto postupovat jinak:
rozdělíme blok *libovolným způsobem* na dva bloky obsahující oba alespoň jeden sloupec (stiskněte knoflík **Krok** v ovladači nazvaném **Konstrukce),**
určíme hodnoty podbloků a
spočteme hodnotu celého bloku následujícím způsobem:

- jestliže levý blok generuje, celý blok generuje také,
- jestliže levý blok absorbuje, celý blok absorbuje také,
- jestliže levý blok přenáší, celý blok má hodnotu rovnou hodnotě pravého bloku.

Pokud levý blok generuje nebo absorbuje, jeho rozhodující sloupec (nejlevější nekomplementární) je také rozhodujícím pro celý blok. Pokud levý blok přenáší, představuje pouze nedůležité přenášející sloupce pro pravý blok.

Hodnoty podbloků, které obsahují více sloupců se určí rekurzivně dalším dělením (používajíce opětovně knoflík **Krok** v ovladači **Konstrukce).**

Při paralelním výpočtu lze hodnoty levého i pravého bloku určovat *současně.*

Rekurzivní výpočet hodnoty bloku lze krokovat knoflíkem **Krok** v ovladači **Výpočet,** který se objeví v okamžiku, kdy je rekurzivní konstrukce stromu výpočtu ukončena.

Knoflíky **Zpět** v obou ovladačích vrací zpět akci knoflíku **Krok** a knoflíky **Dokonči** a **Zruš** umožňují příslušnou akci zcela dokončit nebo vrátit do původního stavu. Knoflík **Široký** umožňuje volit mezi dvěma způsoby kreslení vrcholů stromu rekurze. V prvním je vrchol natažen mezi krajními sloupci bloku, který reprezentuje, v druhém má šířku jednoho sloupce. Druhá metoda bude užívána ve scénách znázorňujících úplnou sčítačku, kde bude zobrazováno několik stromů najednou a široké vrcholy by daly nepřehledné schéma.

Scéna: *Nejhorší dělení*

Pro správnost výpočtu hodnoty bloku je zcela lhostejné, jak jej dělíme do podbloků. Způsob dělení je ale podstatný pro počet paralelních kroků, které je nutno provést k ukončení výpočtu hodnoty. Tato scéna ukazuje nejhorší možný způsob dělení: od celého bloku se vždy oddělí jediný sloupec vpravo nebo vlevo. Schéma výpočtu, které takto dostaneme pak má takovou hloubku, že se sotva vejde na obrazovku a vedlo by k algoritmu výpočetně srovnatelnému s algoritmem ze základní školy.

Vytvořte si široký blok a sledujte jeho strom výpočtu.

Scéna: *Optimální dělení*

Čtenář, který má alespoň základní znalosti v návrhu efektivních algoritmů, jistě ví nebo alespoň tuší, že pro rychlost paralelního výpočtu je optimální dělit blok na dva stejně velké podbloky poloviční velikosti (nebo na dva podbloky lišící se v počtu sloupců o 1 pokud blok sám má lichý počet sloupců a na přesné poloviny rozdělit nejde). Počet paralelních kroků pro výpočet hodnoty je pak $\lceil \log_2 w \rceil$, kde w je počet sloupců bloku a je to to nejlepší, co můžeme dosáhnout.

Zkuste si vytvořit široký blok a sledujte jeho strom výpočtu. Obměňujte blok i sčítance a sledujte konstrukci stromu výpočtu i odpovídající výpočet hodnot bloků.

Scéna: *Hraniční dělení*

Nyní popíšeme jiný způsob dělení, který je pro některé bloky mírně neoptimální, ale má výhody, které se stanou zřejmými později. V této scéně již je nutné předpokládat, že počet šířka operandů je mocninou dvojky (což v současné době v počítačové aritmetice nezpůsobuje prakticky žádná omezení; první mikroprocesory pracovaly se šířkou 8 bitů a záhy byly vystřídány 16-bitovými, nyní končí éra 32-bitových a tato kniha je už psána na 64-bitovém procesoru - vše jsou mocniny 2).

Blok rozdělíme nejprve podle hranice procházející polovinou šířky sčítanců (nikoli polovinou bloku), pak podél hranice procházející první nebo třetí čtvrtinou, pak 1., 3., 5. nebo 7. osminou atd. Pochopitelně jestliže některá hranice blokem neprochází (např. pokud je blok celý v dolní polovině čísel nebo ji přesahuje o jediný sloupec), pak může být odpovídající krok přeskočen.

Blok obsahující všechny sloupce je zajisté dělen optimálně na poloviny. Menší bloky sice mohou být děleny neoptimálně, ale počet úrovní dělení nebude nikdy větší než u všesloupcového bloku. Jelikož se všesloupcový blok v návrhu algoritmu bude vyskytovat, bude paralelní výpočetní složitost dána počtem úrovní rozkladu tohoto největšího bloku a optimalizací rozkladu bloků menších by se počet paralelních kroků výpočtu stejně nemohl zlepšit.

Dělení podle binárních hranic má ale výhodu, že všechny bloky jsou děleny v odpovídajících úrovních stejným způsobem, tedy podle stejných hranic, což se příznivě projeví na tvaru a jednoduchosti výsledného obvodu, viz další scény.

Scéna: *Nejvyšší sloupec*

V této scéně se budeme zabývat otázkou, zda z nejvyššího řádu, tedy ze sloupce úplně vlevo, vychází přenos. Obdobnou otázku budeme muset vyřešit i pro ostatní sloupce, pro levý sloupec je ale nejobtížnější, protože závisí na všech číslicích obou sčítanců.

Pro vyřešení tohoto problému stačí uvažovat blok, obsahující všechny sloupce, tedy prostírající se vpravo od sloupce, který zkoumáme, až ke sloupci, který leží úplně napravo (a tedy máme jistotu, že do něj zprava nemůže vstoupit přenos).

Tento všesloupcový blok nemá nikdy hodnotu '<' a už jsme rozebírali, že pokud generuje, pak z nejvyššího řádu vychází přenos, zatímco pokud absorbuje nebo přenáší, pak z nejvyššího řádu přenos nevychází. Naše otázka se tedy snadno zodpoví určením hodnoty bloku, kterou určujeme na základě dělení bloku podle binárních hranic jak bylo popsáno v předchozí scéně.

Knoflíky ovladače **Konstrukce** si nyní rozviňte obvod pro výpočet hodnoty bloku a knoflíky ovladače **Výpočet** si určujte hodnotu bloku i podbloků vzniklých dělením a tedy i existenci přenosu v nejvyšším řádu pro různé hodnoty sčítanců. Výpočet je krokován po jednotlivých paralelních krocích.

Nyní již je tedy patrna hlavní myšlenka algoritmu: současně s prováděním početních úkonů v pravé části schématu, která zahrnuje méně významné bity, se zpracují levé poloviny sčítanců. Pokud se zjistí, že levá polovina generuje nebo absorbuje, není třeba znát výsledek z pravé části. Nové na algoritmu je ale to, že pokud levá polovina přenáší, pak případný přenos z pravé poloviny sčítanců není nutné "postrkovat" levou částí sloupec po sloupci, jako to činí elementární algoritmus ve své školní podobě vždy a v paralelizované podobě přinejmenším v nejhorším případě, ale přenos "přeskočí" z hranice v polovině sčítanců do nejlevějšího sloupce okamžitě, protože víme, že by k tomu při detailním počítání stejně došlo.

Je vidět, že výpočet je velmi rychlý - počet vrstev obvodu a tedy i počet paralelních kroků potřebných pro výpočet je $\lceil \log_2 n \rceil$, kde n je šířka sčítanců. Obzvláště důležité je to, že schéma výpočtu *nezávisí* na hodnotách sčítanců a výpočet je tedy takto rychlý i v nejhorším případě.

Scéna: *Obecný sloupec*

Podobně jako byl v předchozí scéně určen přenos v nejvyšším řádu, ukážeme si, jak provést určení přenosu v libovolném jiném sloupci. Výběr sloupce pro ilustraci metody provedeme kliknutím na malý bledý čtvereček nad sloupcem.

Jako v předchozí scéně stačí určit hodnotu bloku, určeného vlevo uvažovaným sloupcem a vpravo zasahujícím až na samý pravý okraj sčítanců. Takovýto blok nikdy nemůže dostat zprava přenos z nižšího řádu, protože vpravo od něj již nic není, a v naší klasifikaci nikdy nenabývá hodnoty '<'. Proto z jeho levého sloupce, tedy z našeho uvažovaného sloupce, vychází přenos právě když tento blok generuje.

Při konstrukci si povšimněte, že blok je dělen podle binárních hranic. Jak již bylo uvedeno, je pro nás dostačující aby počet vrstev při dělení bloku nepřekročil počet vrstev, vypočítávajících hodnotu všesloupcového bloku.

Scéna: *Překrývání*

Nyní již víme, jak zkonstruovat úplnou sčítačku: pro každý sloupec zvlášť si sestrojíme obvod, určující zda z tohoto sloupce vychází přenos, způsobem popsaným v předchozích dvou scénách a pak z informací o přenosech určíme v jednom paralelním kroku hodnotu součtu. Všechny obvody pro výpočty přenosů pracují současně.

V této scéně se konečně dozvíte důvod pro dělení bloků podle binárních hranic. Na obrazovce je zobrazen obvod pro výpočet přenosu pro levý sloupec. Klepněte nyní na malý bledý čtvereček nad libovolným jiným sloupcem (výhodně nad sloupcem v levé části schématu). Tím se zobrazí obvod pro výpočet přenosu pro zvolený sloupec. Podle stavu konstrukce (knoflíky ovladače **Konstrukce)** budou obvody v různém stavu vývinu od prostého bloku po úplně dokončený obvod pro výpočet jeho hodnoty.

Nyní již určitě vidíte, proč preferujeme dělení podle binárních hranic. Při tomto způsobu se obvody pro určování přenosu v různých sloupcích podstatně *překrývají*. Není tedy nutné budovat tyto obvody zcela odděleně, což by vedlo k velkému plýtvání hardwarem, ale překrývající se část vytvoříme pouze jednou a její výsledek pak rozvádíme na vstup více než jednoho následného hradla.

Klepáním do čtverečků nad sloupci je možno zobrazovat (a nebo naopak skrývat) obvod pro určení přenosu pro libovolný sloupec a sledovat překrývání těchto obvodů. Je vidět, že překrývání je opravdu velké, například všechny sloupce z levé části mají shodnou pravou část svých obvodů. Nakonec se zjistí, že počet hradel pro *všechny* sloupce není o moc větší než je obvod pro levý, nejsložitější, sloupec.

Je též možno zvolit nebo vypustit několik sloupců najednou: stiskněte myš v bledém čtverci nad některým sloupcem a táhněte myš do strany.

Scéna: *Kompletní sčítačka*

Tato scéna již jen shrnuje, co se v předchozí scéně objevilo při volbě všech sloupců a plně rozvinuté konstrukci, umožňuje ale krokování výpočtu pro různé hodnoty sčítanců.

Teprve na úplné sčítačce je vidět, že je její konstrukce vlastně jednoduchá, je-li formulována rekurzivně: Vytvoříme dva obvody pro výpočet přenosů pro poloviční šířku slova, mírně zobecněné možností přivádět kromě dvou sčítanců ještě zprava přenos za sousedního obvodu. Pravý podobvod již dává polovinu konečných výstupů, výstupy levého podobvodu se ještě všechny zkombinují s nejvýznamnějším výstupem pravé části.

Povšimněte si, že sčítačka pro n-bitové sčítance má $\log_2 n$ vrstev a v důsledku velkého překrývání je v každé vrstvě přesně $n/2$ hradel, které kombinují třístavové hodnoty jednotlivých bloků.

Scéna: *Vrstvená sčítačka*

V kompletní sčítačce ukázané v předchozí scéně byla hradla (vrcholy stromů) uspořádána do vodorovných vrstev, ale jistá propojení mezi (žlutými) hradly přecházela přes několik vrstev. V této scéně je sčítačka doplněna dalšími hradly, vybarvenými růžově, která zaplňují "díry" v obvodu. Růžová hradla neprovádějí žádný výpočet a jsou to ve skutečnosti paměťové elementy, které si mohou zapamatovat jednu třístavovou hodnotu. V každém kroku (kliknutí knoflíku **Krok)** růžové hradlo přečte hodnotu vstupu a zapamatuje si jej. Výstup je vždy roven uchovávané hodnotě.

Růžové prvky jsou velmi podobné tomu, co se v elektronice nazývá klopný obvod typu D; jediný rozdíl je, že si klopný obvod pamatuje jediný bit. Třístavovou veličinu je ale možno uchovávat ve dvou bitech a proto se naše růžové prvky obvykle implementují dvojicí klopných obvodů typu D.

Zkuste si znovu počítat hodnoty přenosů; výpočet probíhá stejně jako v předchozí scéně, ale vstupní hodnoty přicházejí "just in time" a nemusí čekat jako se to stávalo v minulé scéně.

Je vidět, že postup mezivýsledků obvodem je velmi pravidelný, v každém kroku jedna nová vrstva určí své hodnoty. V následujících scénách uvidíme, proč je to užitečné.

Scéna: *Zapomínající sčítačka*

Jak bylo uvedeno v minulé scéně, tok informace v obvodu s růžovými hradly je velmi pravidelný. V každém kroku je aktivní jedna vrstva, která vypočte své hodnoty z hodnot v *předchozí* vrstvě. Protože nás mezivýsledky ve skutečnosti nezajímají a cílem je určit hodnoty poslední vrstvy, je možné mezivýsledky zapomenout, jakmile byly použity následující vrstvou.

Zkuste znova provádět výpočet a uvidíte, že je dostatečné si uchovávat pouze mezivýsledky jediné vrstvy.

Scéna: *Zřetězená sčítačka*

V předchozí scéně jsme viděli, že stačí si pamatovat vždy pouze výsledky jediné vrstvy. Ty vrstvy, jejichž jejichž výsledky již byly použity, "zahálí" a je možno využít k sčítání jiných dvojic sčítanců. Jakmile hodnoty pro jeden pár postoupí o jednu vrstvu dolů, je možné do obvodu vložit nový pár sčítanců, který postupuje obvodem se zpožděním jedné vrstvy.

V této scéně sčítáme plynulý proud dvojic sčítanců a každý pár má svoji barvu, která označuje jemu odpovídající mezivýsledky. Prvky obvodu mají svoji původní barvu pouze dokud k nim nedorazí první mezivýsledky, pak jsou již označovány barvou páru, který právě zpracovávají. Při krokování je jasně vidět, jak informace obvodem postupuje.

Sčítačka typu carry look-ahead je tedy velmi silný nástroj, kromě toho, že jeden pár v obvodu stráví pouze čas úměrný logaritmu počtu bitů a doba trvání výpočtu je zcela nezávislá na jejich hodnotách, mohou do obvodu sčítance vstupovat bezprostředně za sebou s odstupem, který na velikosti sčítanců nezávisí a je dán pouze výpočetním zpožděním použitých výpočetních prvků jedné vrstvy.

Scéna: *Animovaná sčítačka*

Tato scéna přináší inženýrský náhled na implementaci zřetězené sčítačky z předcházející scény. Růžové paměťové prvky jsou nezměněny, pouze jsou nakresleny poněkud menší. Na druhé straně každé žluté výpočetní hradlo je doplněno o jeden růžový paměťová prvek stejného typu jako růžové prvky popsané v předchozích scénách.

Žlutá výpočetní hradla pracují *asynchronně*: jakmile se změní vstupní hodnoty, odpovídajícím způsobem se ihned změní i výstup. Přesněji řečeno, změna se objeví na výstupu po jistém časovém okamžiku, zpoždění, které může být *různé* pro různá hradla.

Pokud bychom zpracovávali pouze jeden pár sčítanců, růžové paměťové prvky by nebyly třeba, ale rychlost postupu informace by byla v různých částech obvodu jiná. To by ale v případě zřetězeného zpracování více párů sčítanců vedlo k tomu, že by nebylo možné hodnoty odpovídající různým párům na výstupu obvodu odlišit.

Z tohoto důvodu musíme používat *synchronní* obvod a synchronizace je dosahována růžovými paměťovými prvky ve všech pozicích obvodu. Výstup paměťového obvodu je roven uchovávané hodnotě; po většinu doby prvek ignoruje vstupní hodnotu. Paměťové prvky však dostávají hodinový signál (jeho přivádění k prvkům není znázorněno). Jakmile paměťový prvek dostane hodinový signál (v reálných systémech například vzestupná hrana hodinového signálu, u našeho appletu kliknutí knoflíku **Krok),** prvek přečte vstupní hodnotu a nahradí jí dříve uchovávanou hodnotu. Po této mžikové události se zapamatovaná hodnota dále nemění až do dalšího hodinového impulzu.

Zkuste si výpočet s použitím knoflíku **Krok;** v okamžiku hodinového impulzu (kliknutí) všechny paměťové prvky najednou nahradí původní uchovávanou hodnotu hodnotou, která byla na vstupu. Animace ukazuje, jak informace postupuje podél vodičů, připojujících výstupy paměťových prvků se vstupy následujících hradel, kterými mohou být výpočetní hradla nebo další paměťové prvky.

Jakmile informace dorazí na výpočetní hradlo, je zpracována a s jistým zpožděním se výsledek objeví na jeho výstupu (zpoždění jsou různá pro různá hradla - rozdíly jsou úmyslně zvýrazněny), což se projeví m.j. změnou barvy hradla.

Barvy hradel a signálů reflektují barvu odpovídajícího páru sčítanců, ke kterému se vztahují. Je tedy snadné sledovat postup informace obvodem.

Jistě je vám zřejmé, že hodinový kmitočet musí být takový, aby doba mezi dvěma po sobě následujícími hodinovými pulzy byla dostatečná pro přenos signálu mezi hradly a jeho zpracování. Náš applet je silně synchronizován a nepřijímá kliknutí, dokud neprovede všechny nutné operace, ale v případě reálného obvodu by příliš rychlé hodiny vedly k chybným výpočtům.

Scéna: *Číslování*

Tato scéna již není součástí vlastní procházky appletem, nicméně si ukážeme jistou magii čísel.

Sloupce sčítačky jsou očíslovány zprava doleva od 0 do $n-1$, kde n je bitová šířka sčítanců, v úvodním případě $n = 32$. Vrstvy hradel jsou očíslovány shora dolů od 0 do $\log_2 n - 1$ (pamatujte, že n je mocnina 2 a proto je $\log_2 n$ celé číslo). Na úvodním obrázku je $\log_2 32 = 5$. Vrstva operandů je očíslována jako -1.

Sčítačka je znázorněna v úplné podobě se žlutými výpočetními hradly se dvěma vstupy a vloženými růžovými paměťovým obvody. Hradla jsou očíslována čísly, které nemají žádnou souvislost s operandy nebo mezivýsledky, ale popisují typ hradla: 1 je žluté výpočetní hradlo a 0 je růžové paměťové hradlo.

Za této situace se podívejte na posloupnost 0 a 1 v hradlech libovolného sloupce a zjistíte, že představuje binární zápis čísla sloupce (s nejvýznamnějším bitem dole a nejméně významným nahoře ve vrstvě 0). Ve vrstvě očíslované i je číslice, která v binárním zápisu vystupuje u mocniny 2^i.

Stiskněte nyní myš nad libovolným hradlem. Odpovídající obdélník zčervená a zčervenají také obdélníky hradel, na jejichž hodnotách závisí hodnota stisknutého hradla. Zčervenalé obdélníky ve vrstvě operandů (očíslované -1) vytvářejí interval, jehož levý konec je ve sloupci, ve kterém se nachází stisknuté hradlo, zatímco pravý konec je ve sloupci, jehož pořadové číslo se dostane tak, že ve sloupci obsahujícím stisknuté hradlo nahradíme nulou číslo v stisknutém hradlu a čísla ve všech obdélnících nad stisknutým hradlem.

Stiskněte např. hradlo v sloupci 29 ve vrstvě 3. Jelikož 29 je binárně 11101, pak nahrazením číslic ve vrstvách 3, 2, 1, 0 nulami dostaneme 10000 neboli dekadicky 16 a uvidíte, že při stisknutí zmíněného hradla zčervenají operandy od sloupce 29 až po sloupec 16.

U žlutého hradla jsou oba vstupy připojeny na výstupy hradel v předchozí vrstvě. Jeden vstup je připojen na hradlo ve stejném sloupci, druhý na hradlo ve sloupci, jehož pořadové číslo se dostane tak, že se v sloupci se stisknutým hradlem nahradí nulou všechny číslice nad stisknutým hradlem, a pak se od vzniklého čísla odečte 1.

Stiskneme-li ještě jednou hradlo ve vrstvě 3 a sloupci 29 neboli binárně 11101, pak po náhradě nulou ve vrstvách 2, 1, 0 dostaneme 11000, neboli 24 a

vidíme, že pravý vstup hradla je napojen skutečně do sloupce 23 = 24-1. Levý vstup totiž závisí na číslicích operandů od sloupce 29 do $11000_2 = 24$ a proto pravý vstup hradla zpracovává hodnoty od následujícího sloupce vpravo, tedy od sloupce 24-1=23.

Kapitola 19

Diskrétní Fourierova transformace

Scéna: *Kreslení funkce*

Cílem prvních několika scén je pouze naučit se applet ovládat. V první scéně je znázorněno okno, ve kterém je graf funkce. Toto okno budeme nazývat *funkční okno*. Graf je možno myší překreslit; vyzkoušejte si to.

Scéna: *Zadávání funkce vzorcem*

Funkci, jejíž graf je znázorněn v okně, je možno zadat i vzorcem, zapsaným do pole pro zadávání vzorce. Jako příklad je možno uvést funkce `sin(2*PI*x)`, `cos(2*PI*x)+exp(x-1)`, `sgn(x-0.5)`, `abs(x-0.5)+log(1+x)` (další použitelné funkce jsou tg, cotg, ln, sqrt). Podrobně je syntax popsána v okénku, které se objeví po stisknutí knoflíku **Syntax.** Pro jednoduchost je kalkulátor upraven tak, že pokud je výraz pro některé x nedefinován (například po dělení nulou, logaritmus nebo odmocnina záporného čísla a podobně), nahradí jej nulou.

V řadě příkladů bude pro lepší zobrazení grafu funkce nutné změnit měřítko x-ové nebo y-ové osy změnou čísel v polích $x1$, $x2$ (levý a pravý okraj okna na ose x), $y1$, $y2$ (dolní a horní okraj okna na ose y).

Scéna: *Sinusoidy*

Nejdůležitější funkce ve Fourierově analýze jsou funkce odvozené z funkce sinus, z nichž některé si zobrazíme v této scéně. I když si myslím, že vlastnosti těchto funkcí dobře znáte, nebude na škodu si scénu rychle projít.

Základní sinusoidy jsou funkce $\sin(2\pi x)$ a $\cos(2\pi x)$, které jsou periodické s periodou 1. Proto je funkční okno upraveno na zobrazování hodnot funkcí pro $x \in [0, 1]$.

Na obrazovce je zobrazeno další okno, které budeme z důvodů, které se stanou zřejmými později, nazývat *spektrální okno* a které budeme zatím používat velmi omezeným způsobem:

V dolní části spektrálního okna je řada checkboxů, jeden z nich je zvolen. Klepnutím na jiný checkbox jej zvolíte a tím se automaticky vypne checkbox, který byl zvolen před tím. Zvolený checkbox udává charakter funkce, která bude zobrazena ve funkčním okně.

Na začátku je zvolen modrý checkbox a ve funkčním okně se zobrazuje často používaná konstantní funkce $f(x) = 1$.

Zvolte nyní červený checkbox s označením 1. Zobrazí se sinusovka $f(x) = \sin(2\pi x)$ a je-li zvolen zelený checkbox s označením 1, zobrazí se kosinusovka $f(x) = \cos(2\pi x)$. Obě tyto funkce mají periodu 1 odpovídající šířce funkčního okna.

Nyní zvolte nejprve červený checkbox a potom zelený checkbox označený 2. Zobrazí se sinusovka $f(x) = \sin(2 \cdot 2\pi x)$ respektive kosinusovka $f(x) = \cos(2 \cdot 2\pi x)$. Jsou podobné dvěma předcházejícím funkcím, ale jejich perioda je poloviční, neboli jejich frekvence je dvojnásobná. Plná šíře funkčního okna zobrazuje dvě periody těchto funkcí.

Zbývající checkboxy ve spektrálním okně slouží pro zobrazení sinusovek a kosinusovek vyšších frekvencí. Klepnutím na červený nebo zelený čtvereček s označením k se zobrazí sinusovka $f(x) = \sin(k \cdot 2\pi x)$ respektive kosinusovka $f(x) = \cos(k \cdot 2\pi x)$. Opět mají podobný průběh jako základní funkce $\sin(2\pi x)$ a $\cos(2\pi x)$, ale periodu k-krát menší neboli frekvenci k-krát větší.

Výška červeného, zeleného nebo modrého sloupce ve spektrálním okně může být měněna zachycením za jeho horní okraj (nebo dolní okraj, vyčníváli pod osu x) a tažením; tím se mění amplituda zobrazované funkce. Zkuste si znovu volit sinusovky a kosinusovky různých frekvencí (a konstantní funkci) a měnit jejich amplitudy.

Scéna: *Lineární kombinace sinusoid*

Scéna je jednoduché, ale důležité cvičení na používání spektrálního okna. V předchozí scéně bylo možné volit jedinou sinovou funkci a měnit její amplitudu. V této scéně je možné zvolit několik sinových funkcí najednou (třeba i všechny) a ve funkčním oknu se zobrazí jejich *součet*.

V řadě případů bude asi potřeba si pro lepší zobrazení funkce změnit měřítko osy y (změny na ose x se v této scéně provádět nedají).

Ve skutečnosti jsou v této scéně na začátku zvoleny *všechny* čtverečky ve spektrálním okně, ale amplitudy jsou až na jednu voleny rovny 0 (což je ekvivalentní nezvolení čtverečku).

Měňte amplitudy v různých sloupcích spektrálního okna jako při mixování zvuku na mixážním pultu a pozorujte, jak se mění odpovídající součtová funkce ve funkčním okně.

Libovolný sloupec ve spektrálním okně se dá "vypnout" kliknutím myší; odpovídající (ko)sinusovka pak nepřispívá k součtu ve spektrálním okně. Vypnutí má stejný účinek jako snížení amplitudy na 0.

Velmi doporučuji u této scény zůstat co nejdéle; volbou různých množin funkcí a především volbou jejich různých amplitud je možno vytvořit velmi bohatý soubor odvozených funkcí a pro další porozumění diskrétní Fourierově transformaci velmi pomůže, budete-li jich umět vytvořit co nejvíce, včetně funkcí velmi kuriózních a "divoce" se měnících.

Scéna: *Vzorkování spojité funkce*

Zapomeňme na chvíli na omezené rozlišení obrazovky a představujme si, že osa x ve funkčním okně je kontinuum *všech* reálných čísel. Pro popis funkce nakreslené nad takovou osou bychom potřebovali nekonečné množství údajů a pokud by funkce například představovala akustický nebo elektrický signál, musela by se každá hodnota odečíst v nekonečně malém časovém okamžiku, což pochopitelně není možné.

Při digitálním zpracování signálů se proto signál obvykle *vzorkuje*, což znamená, že se odečítá je v některých okamžicích, které následují po sobě s konstantními časovými rozestupy. Pokud je perioda signálu 1 a během ní provedeme n vzorků, pak je odečítáme v okamžicích $x_0, \ldots, x_{n-1}$, kde $x_\ell = \ell/n$ pro $\ell = 0, \ldots, n-1$. Protože funkce je periodická, pro $x_n = n/n = 1$ bychom již dostali stejnou hodnotu jako pro $x_0 = 0$.

Vzorkovací body jsou ve funkčním okně znázorněny svislými čárami a hodnoty funkce v těchto bodech jsou zvýrazněny malými kolečky. Počet vzorkovacích bodů lze změnit na ovladači (pole **Vzorků** N**),** ale měl by být malé sudé číslo. Vzhledem k pravidelnosti jejich rozmístění je tento údaj dostatečný pro určení jejich polohy.

Scéna: *Spektrální analýza*

Ve funkčním okně jsou zobrazeny dvě funkce. Funkce znázorněná černě (na začátku sinusovka) může být překreslena myší nebo změněna zadáním vzorce nové funkce, jak bylo popsáno v prvních dvou scénách této kapitoly. Je také možno měnit frekvenci vzorkování a měřítka os.

Druhá funkce ve funkčním okně, znázorněná červeně, odpovídá amplitudám ve spektrálním okně způsobem, který byl popsán ve scéně o lineárních kombinacích sinusovek. Ve funkčním okně ji nelze překreslit.

Pro danou černou funkci jsou amplitudy ve spektrálním okně určovány Algovizí tak, aby jim odpovídající červená funkce se shodovala s černou *ve všech* vzorkovacích bodech. Je zřejmé, že nelze volit amplitudy tak, aby se

červená funkce shodovala s černou všude; černý graf má daleko větší počet stupňů volnosti než soubor amplitud ve spektrálním okně, ale hlavní výsledek diskrétní Fourierovy analýzy říká, že shodu ve vzorkovacích bodech je možno zajistit pro *každou* periodickou černou funkci.

Soubor amplitud ve spektrálním okně se nazývá *spektrum* černé funkce. To je také důvod názvu spektrálního okna, kde jsou tyto hodnoty zobrazovány.

Měňte černou funkci a pozorujte její spektrum, vypočtené Algovizí a odpovídající červenou funkci. Zkuste si hlavně příklady funkcí, které se prudce (skokem) mění. U nich dochází k typickému průběhu rozdílové funkce, který je dobře znám z chování elektrických obvodů: pokud se hodnota černá funkce skokem změní a pak zůstává konstantní, červená funkce v reakci na skok "překmitne" za hodnotu černé funkce a pak se k její nové hodnotě přibližuje způsobem připomínajícím tlumené kmity kolem nové hodnoty černé funkce s tím, že ke shodě obou funkcí dochází ve vzorkovacích bodech. Analogie však není úplná: chování červené funkce je "časově reverzibilní" a rozdílová funkce se "rozkmitá" i v přípravě na skok dříve než k němu dojde.

Scéna: *Hledání spektra*

Tato scéna je nejdůležitější ze všech a přináší mimořádně obtížné cvičení. Je třeba udělat to, co dělala v předchozí scéně Algovize: aproximovat černou funkci, která je nakreslena ve funkčním oknu, ve vzorkovacích bodech červenou funkcí. Výšky sloupečků ve spektrálním okně, představující amplitudy jednotlivých harmonických, je možno měnit tahem myši, jak to již bylo vysvětleno dříve a cílem je dosáhnout přesné shody černé a červené funkce ve vzorkovacích bodech. Tato scéna je další, ve které byste měli strávit hodně času a zkoušet aproximace nejrůznějších černých funkcí.

Tak například změna konstantní složky posouvá červenou funkci nahoru a dolů a je ji třeba nastavit tak, aby střední hodnota černé i červené funkce přes celou periodu byla stejná. Základní sinová složka posouvá hodnoty červené funkce v první čtvrtině periody nahoru a hodnoty v 3. čtvrtině periody o stejnou hodnotu dolů, zatímco na začátku a konci periody a v její polovině jsou hodnoty červené funkce na této harmonické nezávislé. Na druhé straně základní kosinová složka nechává hodnoty v 1. a 3. čtvrtině stejné a mění hodnoty na začátku periody a v její polovině o stejnou velikost v opačných směrech. Vyšší harmonické, pokud mají menší amplitudu, umožňují dosáhnout jemného zvlnění průběhu.

Vzhledem k obtížnosti úlohy by bylo překvapivé, kdyby začátečník našel správně úplné spektrum. Je proto možno požádat o pomoc - nastavení správné amplitudy pro jednu složku spektra klepnutím na čtverec označený '?' pod příslušným sloupcem ve spektrálním okně.

Doporučuji začít hledání spektra od konstantní složky a pak nejdříve nízkých harmonických a pak teprve vyšších a (alespoň ze začátku) požádat po

pokusu o nastavení amplitudy Algovizi o nastavení správné hodnoty dříve než přejdete k další složce.

Důležitým, ale obtížným úkolem je umět vytvořit spektrum černé funkce, která se objeví při vstupu do scény a mění svou hodnotu skokem v polovině funkčního okna (a na jeho konci se vrací zpět).

Scéna: *Spektrální komprese*

Tato scéna je stejná jako scéna Spektrální analýza, liší se ale poněkud v ovládání a i svým cílem. Amplitudy složek spektra nelze měnit, zůstávají takové, jaké je Algovize vypočetla pro danou černou funkci. Ve spektrálním oknu je ale možno některé sloupce, tj. složky spektra, deaktivovat klepnutím na čtvereček pod sloupcem. Vypnuté sloupce pak nepřispívají do lineární kombinace, která je uvedena v popisu předchozí scény. To pochopitelně vede k tomu, že červená lineární kombinace sinusoid se již výchozí černé funkci nemusí rovnat a obvykle nerovná ani ve vzorkovacích bodech. Je ale možno pozorovat, že pokud se anulují vysoké harmonické, tedy sloupce určující amplitudy sinusoid s vysokou frekvencí, zůstává celkový charakter funkce změněn jen málo.

Pokud například funkce popisuje akustický signál, pak odebráním vysokých harmonických se může mírně změnit barva tónu, ale signál není zásadně změněn; například řeč je i nadále srozumitelná a hudba působí stejným dojmem, i když poněkud ploše.

Podobného principu se používá i u známé obrazové komprese JPEG, kde se provede spektrální rozklad dat, méně důležité harmonické se potlačí nebo omezí a pak se z upraveného spektra zrekonstruuje upravený obraz, který je mnohdy velmi obtížné odlišit od původního. Přitom ale je množství informace obsažené ve spektru sníženo a proto dochází k datové kompresi. Na druhé straně je ovšem známo, že v některých případech, například v místě, kde se stýkají dvě jasné a monochromatické a barevně rozlišené plochy, mohou vznikat artefakty, které jsou způsobeny podobnými jevy jako jsou překmity, které byly popsány výše.

Standardní JPEG je sice založen na kosinové transformaci, která je poněkud odlišná od zde popsaného způsobu vytváření spektra, ale rozdíl je spíše formální a zaměřený na snadnější provádění výpočtů a základní podstata obou transformací je stejná. Zhruba je možno říci, že kosinová transformace je úprava diskrétní Fourierovy transformace (viz závěr) tak, aby nebylo nutno používat komplexních čísel.

Závěr

Ve scéně Spektrální analýza jsme ukazovali, jak aproximovat černou periodickou funkci f s periodou 1 v n pravidelně rozmístěných bodech $x_k = k/n$,

$k = 0, 1, \ldots, n-1$. Pro $k = 0, 1, \ldots, n-1$ označíme jako A_k hodnotu funkce f v bodě x_k.

Ve scénách jsme přitom předpokládali, že n bylo sudé číslo. Označme $m = n/2$.

Základní poznatek který zde byl ilustrován je ten, že existují reálná čísla $\alpha_1, \ldots, \alpha_{m-1}$, $\beta_1, \ldots, \beta_m$ a δ taková, že

$$A_k = \sum_{\ell=1}^{m-1} \alpha_\ell \sin(2\pi\ell x_k) + \sum_{\ell=1}^{m} \beta_\ell \cos(2\pi\ell x_k) + \delta, \tag{1}$$

pro $k = 0, \ldots, n-1$. Jistě vám je jasné, proč se sinusovky sčítají jen do $\ell = m-1$, kdežto kosinusovky do $\ell = m$: funkce $\sin(2\pi m x)$ je totiž ve všech bodech $x_0, x_1, \ldots, x_{n-1}$ rovna nule a tudíž její přidání do vzorce by k ničemu novému nevedlo. Platí totiž, že

$$\sin(2\pi m x_\ell) = \sin(2\pi m\ell/n) = \sin(\pi\ell) = 0 \tag{2}$$

pro každé celé číslo ℓ.

Existence rozkladu (1) znamená, že se ve scéně Spektrální analýza ve vzorkovaných okamžicích, vyznačených svislými šedivými čarami, červenou funkcí (která představuje lineární kombinaci konstanty, sinusovek a kosinusovek) vždy přesně trefíme do hodnot $A_0, A_1, \ldots, A_{n-1}$ černé funkce. Jak jsme viděli, pro všechny černé funkce se to vždy podařilo. Korektní matematický důkaz ukáži za chvíli.

Nejprve ale upravíme výše uvedený vzorec standardním způsobem za pomoci exponenciálních funkcí s komplexními koeficienty tak, aby se nám s ním lépe pracovalo. S komplexními čísly budeme od této chvíle pracovat důsledně, rozklad (1) budeme hledat i pro případ, kdy čísla A_k jsou komplexní.

Ukážeme nyní, že pokud jsou dána komplexní čísla $A_0, \ldots, A_{n-1}$, pak komplexní čísla $\alpha_1, \ldots, \alpha_{m-1}$, $\beta_1, \ldots, \beta_m$ a δ taková, že platí (1) existují právě když existují komplexní čísla $a_{-m+1}, \ldots, a_0, \ldots, a_m$ taková, že

$$A_k = \sum_{\ell=-m+1}^{m} a_\ell \omega^{\ell k} \qquad \text{pro } k = 0, \ldots, n-1, \tag{3}$$

a to je právě když existují komplexní čísla $a_0, \ldots, a_{n-1}$ taková, že

$$A_k = \sum_{\ell=0}^{n-1} a_\ell \omega^{\ell k} \qquad \text{pro } k = 0, \ldots, n-1, \tag{4}$$

kde jsme označili pro zjednodušení zápisu $\omega = e^{2\pi i/n}$.

Je totiž známo, že

$$e^{ix} = \cos x + i \sin x, \tag{5}$$

kde $i = \sqrt{-1}$ je komplexní jednotka. Z toho ihned plyne, že

$$\sin x = \frac{e^{ix} - e^{-ix}}{2i} \quad \text{a} \quad \cos x = \frac{e^{ix} + e^{-ix}}{2}. \tag{6}$$

Kromě toho bylo již výše uvedeno, že v bodech $x_0, x_1, \ldots, x_{n-1}$ je funkce $\sin(2\pi m x)$ rovna nule, takže pak $\cos(2\pi m x_\ell) = e^{2\pi i m x_\ell}$ pro $\ell = 0, 1, \ldots, n-1$.

Platí-li (1), pak dosadíme-li ze vzorců (6) do rozvoje (1), vidíme, že A_k jsou lineárními kombinacemi hodnot funkcí tvaru $e^{2\pi i \ell x}$, $\ell = -m+1, \ldots, 0, \ldots, m$ v bodech $x_0, x_1, \ldots, x_{n-1}$, takže pro nějaká reálná čísla $a_{-m+1}, \ldots, a_0, \ldots, a_m$ opravdu platí (3). Naopak je-li dán rozklad (3) a za exponenciální funkce s imaginárními exponenty dosadíme ze vzorce (5), dostaneme s uvážením nulovosti funkce $\sin(2\pi m x)$ ve vzorkovacích bodech ihned rozklad (1).

Ekvivalence (3) a (4) plyne z toho, že pro libovolná celá čísla ℓ a k platí

$$\omega^{(\ell+n)k} = \omega^\ell \omega^{nk} = \omega^\ell \left(\omega^n\right)^k = \omega^\ell 1^k = \omega^\ell, \tag{7}$$

protože

$$\omega^n = \left(e^{2\pi i/n}\right)^n = e^{2\pi i} = \cos(2\pi) + i \sin(2\pi) = 1. \tag{8}$$

Z toho plyne, že

$$\omega^{-(m-1)k} = \omega^{(m+1)k}, \ldots, \omega^{-k} = \omega^{(n-1)k} \tag{9}$$

pro libovolné celé číslo k a tedy stačí položit $a_{m+1} = a_{-m+1}, \ldots, a_{n-1} = a_{-1}$.

Vztah (3) nebo (4) se nazývá *Diskrétní Fourierova Transformace*. Formulace (3) je výhodnější při vysvětlování motivace, ale v dalším budeme za její definici považovat vztah (4). Způsobem jejího výpočtu se budeme zabývat v následující kapitole.

Nyní již dokážeme existenci rozkladů (1), (3) a (4), což znamená, že pro libovolnou posloupnost $[A_0, \ldots, A_{n-1}]$ komplexních čísel existuje právě jedna posloupnost $[a_0, \ldots, a_{n-1}]$ reálných čísel taková, že pro ně platí vztah (4). Pro $\ell = 0, \ldots, n-1$ si jako $\mathbf{b}_\ell$ označíme vektor $[\omega^{0\cdot\ell}, \omega^{0\cdot\ell}, \ldots, \omega^{(n-1)\cdot\ell}]$. Vektory $[A_0, \ldots, A_{n-1}]$ vytvářejí n-dimenzionální vektorový prostor a stačí dokázat, že vektory $\mathbf{b}_0, \ldots, \mathbf{b}_{n-1}$ v něm vytvářejí bázi. Jelikož je jich tolik, kolik je dimenze prostoru, je dostatečné podle jedné ze základních vět lineární algebry, Steinitzovy věty o výměně, dokázat že jsou lineárně nezávislé a k tomu zase stačí dokázat, že jsou na sebe kolmé, neboli že jejich skalární součin je roven nule.

Ve vektorovém prostoru nad tělesem komplexních čísel je skalární součin dvou vektorů $[\xi_0, \ldots, \xi_{n-1}]$ a $[\nu_0, \ldots, \nu_{n-1}]$ definován jako $\xi_0\overline{\nu_0} + \cdots + \xi_{n-1}\overline{\nu_{n-1}}$, kde $\overline{x}$ znamená číslo komplexně sdružené k x. Z vzorce (5) plyne, že komplexně sdružené hodnota k e^{ix} je e^{-ix} a tedy komplexně sdružená hodnota k ω^x je ω^{-x}.

Jak jsme ukázali výše, je $\omega^n = 1$ a je zřejmé, že čísla $\omega, \omega^2, \ldots, \omega^{n-1}$ jsou všechna různá od 1. Pokud tedy jsou $0 \leq r, s < n$ dvě celá čísla taková, že $r \neq s$, pak označíme-li $q = \omega^{r-s}$, je skalární součin vektorů $\mathbf{b}_r$ a $\mathbf{b}_s$ roven

$$\sum_{k=0}^{n-1} \omega^{kr}\omega^{-ks} = \sum_{k=0}^{n-1} \omega^{k(r-s)} = 1 + q + q^2 + \cdots + q^{n-1} = \frac{q^n - 1}{q - 1} = 0, \quad (10)$$

protože je $q \neq 1$ a ze vzorce (8) plyne, že $q^n = \omega^{n(r-s)} = (\omega^n)^{r-s} = 1^{r-s} = 1$.

Nakonec se podíváme na poslední důležitý pojem - na inverzní diskrétní Fourierovu transformaci. Je principiálně jednoduché spočítat pomocí vztahu (4) hodnoty A_k, jsou-li dány hodnoty a_ℓ, což vlastně znamená ze znalosti složek spektra spočítat hodnoty studované funkce ve vzorkovacích bodech. Opačná úloha, tedy spočítat hodnoty a_ℓ tak aby platilo (4), jsou-li dány hodnoty A_k, se může zdát naopak velmi složitá - je to určení spektra dané funkce. Ukážeme, že ale tento problém - spočítání na inverzní diskrétní Fourierovy transformace - není o nic složitější, než počítání přímé transformace.

Již jsme si ukázali (vztah (10)), že výraz

$$P_{r,s} = \omega^{0\cdot r}\omega^{-0\cdot s} + \cdots + \omega^{(n-1)\cdot r}\omega^{-(n-1)\cdot s}$$

je roven nule pro $0 \leq r, s < n$ takové, že $r \neq s$. Jestliže je $r = s$, pak $\omega^{\ell\cdot r}\omega^{-\ell\cdot s} = \omega^{\ell\cdot(r-s)} = \omega^0 = 1$ pro každé ℓ a tedy $P_{r,r} = n$.

Podívejme se nyní na vztah (4) jako na násobení vektoru maticí. Označíme-li totiž jako A sloupcový vektor se složkami $A_0, \ldots, A_{n-1}$ a jako a sloupcový vektor se složkami $a_0, \ldots, a_{n-1}$ a dále jako Ω matici, která má v k-tém řádku a ℓ-tém sloupci číslo $\omega^{k\cdot\ell}$, pak vztah (4) znamená, že $A = \Omega a$. Naším cílem je najít matici Ω^{-1} inverzní k Ω, protože pak bude $a = \Omega^{-1}A$.

Ukážeme nyní, že je-li M matice, která má v k-tém řádku a ℓ-tém sloupci číslo $\omega^{-k\cdot\ell}$, pak $\Omega^{-1} = \frac{1}{n}M$. Snadno se totiž zjistí, že v k-tém řádku a ℓ-tém sloupci matice ΩM je číslo $\omega^{0\cdot r}\omega^{-0\cdot s} + \cdots + \omega^{(n-1)\cdot r}\omega^{-(n-1)\cdot s} = P_{k,\ell}$ a z výše uvedeného již okamžitě plyne, že ΩM je n-násobek jednotkové matice, tedy $\frac{1}{n}M$ je opravdu inverzní k Ω.

Z toho nakonec plyne, že inverzní diskrétní Fourierova transformace vektoru $[a_0, \ldots, a_{n-1}]$, tedy úloha nalézt vektor $[A_0, \ldots, A_{n-1}]$ tak aby platilo (4), je vyřešena následujícím vzorcem:

$$a_\ell = \frac{1}{n}\sum_{k=0}^{n-1} A_k\omega^{-k\ell} \qquad \text{pro } k = \ell, \ldots, n-1, \quad (11).$$

Uváží-li se, že n-tá mocnina čísla ω^{-1} je také 1, protože $\left(\omega^{-1}\right)^n = \omega^{-n} = 1/\omega^n = 1/1 = 1$, pak až na dělení číslem n u inverzní transformace jsou výpočty přímé a inverzní Fourierovy transformace prakticky stejné.

Kapitola 20

Rychlá Fourierova transformace

Scéna: *Rychlá Fourierova transformace*

Pokud neznáte definici a aplikace diskrétní Fourierovy transformace a přeskočili jste předchozí kapitolu, kde se o nich jedná, doporučuji Vám abyste si ji nejprve prostudovali. Není to sice nutné pro pochopení algoritmu rychlé Fourierovy transformace (FFT - Fast Fourier Transform), ale bylo by vhodné, abyste věděli, proč je užitečné umět počítat podivně vyhlížející transformaci a hlavně proč je třeba ji umět počítat extrémně rychle.

Zde pouze uvedeme, že úloha je následující: je dán vektor $(a_0, \ldots, a_{n-1})$ a je třeba vypočítat vektor $(A_0, \ldots, A_{n-1})$ definovaný takto

$$A_k = \sum_{j=0}^{n-1} \omega^{k \cdot j} a_j \quad \text{pro } k = 0, \ldots, n-1,$$

kde ω je číslo takové, že $\omega^n = 1$. Příkladem takového čísla je

$$\omega = e^{2\pi i/n}, \qquad \text{nebo} \qquad \omega = e^{-2\pi i/n},$$

kde $i = \sqrt{-1}$ je imaginární jednotka, takže výše ukázaný vzorec zahrnuje vztahy (4) a (11) z předchozí kapitoly (až na dělení číslem n v (11)), tedy přímou i inverzní Diskrétní Fourierovu Transformaci.

Pro FFT je nutné předpokládat, že dimenze n vektorů je mocnina 2, což z praktického hlediska není příliš velké omezení, šířka představovaná mocninou 2 je v počítačích běžná, ba dá se říci je téměř pravidlem.

V této scéně budeme zápis diskrétní Fourierova transformace, vyplývající z definice, upravovat až z něho dostaneme schéma rychlé Fourierova transformace. Postup spočívá v opakovaném provádění následující posloupnosti kroků, které budete provádět knoflíkem **Krok:**

Krok: *Zadání*

První krok ukazuje definici diskrétní Fourierovy transformace v rozvinuté maticové podobě a vypínatelně i jako vzorec. Dimenze vektoru je v rozumné míře volitelná (jako mocnina 2) na ovladači, jako výchozí je voleno $n = 8$, která již není triviální, ale stále umožňuje používat dostatečně velké fonty. ◇

Krok: *Formální modifikace zadání*

Zatímco faktor ω^0 (rovný 1) a exponent u ω^1 nebyly v předchozím kroku zobrazovány, nyní jsou uvedeny v plné podobě. ◇

Krok: *Rozdělení sloupců*

Sloupce se rozdělily na sudé a liché, což je znázorněno barvou. ◇

Krok: *Přemístění sloupců*

Sloupce se přemístí tak, aby (původně) liché byly separovány od (původně) sudých. ◇

Krok: *Vytknutí mocniny ω*

V k-tém řádku zahrnují všechny členy v pravé polovině mocninu ω^k; její vytknutí je nejprve naznačeno a potom provedeno. ◇

Krok: *Klíčový krok*

Nyní přicházíme k základní úpravě, zahrnující fundamentální pozorování sloužící pro úpravu vzorců, jako jediná využívá vztah $\omega^n = 1$ a způsobuje, že je FFT tak rychlý algoritmus.

Podíváme-li se na libovolný člen v horní polovině schématu a člen, který se nachází přesně $n/2$ řádků pod ním, pak exponenty ω v těchto dvou členech se liší o celočíselný násobek čísla n. Mocnina ω je totiž tvaru $\omega^{s\cdot k}$, kde k je číslo řádku, ve kterém se člen nachází a jestliže člen byl původně v S-tém sloupci, pak buď $s = S$, pokud S bylo sudé nebo $s = S - 1$, pokud S bylo liché. Číslo s je tedy v každém případě sudé a rozdíl indexů řádků, ve kterých se námi uvažované členy nacházejí, je $n/2$. Rozdíl exponentů je tedy sudý násobek $n/2$ neboli celočíselný násobek n a stačí si uvědomit, že $\omega^n = 1$, neboli

$$\omega^{s\cdot k} = \omega^{s\cdot k} \cdot 1^{s/2} = \omega^{s\cdot k} \cdot [\omega^n]^{s/2} = \omega^{s\cdot k} \cdot \omega^{sn/2} = \omega^{s(k+n/2)},$$

kde vlevo je mocnina ω z horního členu a vpravo je mocnina ω z dolního členu.

V dolní části se proto exponenty čísla ω změní tak, aby se shodovaly s exponenty ω ve výrazu nacházejícím se o $n/2$ řádků výše. Operace je nejprve naznačena a poté provedena. Porovnejte si hodnoty exponentů pro některé konkrétní dvojice členů před úpravou i po ní. ⋄

Krok: *Úprava zápisu*

Naším cílem je nejen ukázat jak naprogramovat algoritmus FFT na obvyklém počítači, ale také předvést layout jednoduchého a elegantního obvodu, který se používá pro její výpočet. Zatímco dosud byl zápis schématu čistě matematický a každý řádek byl tvaru $(\ldots) + \omega^k(\ldots)$, nyní se spočtou výrazy v závorkách, popsané stále jako matematický výraz, ale tyto dvě hodnoty se přivedou na vstup dvou hradel, která nejprve druhý vstup vynásobí číslem ω^ka pak se obě hodnoty sečtou. ⋄

Krok: *Porovnání SV a JV matice*

Předchozími úpravami se schéma výpočtu změnilo tak, že přirozeným způsobem zahrnuje čtyři maticové bloky řádu $n/2 \times n/2$: dva jsou horní (severní) a dva dolní (jižní), ale současně je také možno dva označit za levé (západní) a dva za pravé (východní). Jak již bylo řečeno, SV a JV bloky jsou po provedených úpravách totožné; totéž platí i pro SZ a JZ bloky. Matice z první dvojice jsou graficky zvýrazněny; ověřte si jejich shodnost. ⋄

Krok: *Překrytí SV a JV matice*

Jak již bylo uvedeno v předchozím kroku, SV a JV bloky jsou shodné a to bylo nyní zdůrazněno jejich překrytím. Poznamenejme, že applet i nadále vykresluje na obrazovku *oba* bloky. To že je vidět jen jeden podtrhuje fakt, že se bloky vůbec neliší. ⋄

Krok: *Porovnání SZ a JZ matice*

Jak již bylo řečeno, SZ a JZ blok jsou také shodné. Pro usnadnění porovnání jsou nyní graficky zvýrazněny. ⋄

Krok: *Překrytí SZ a JZ matice*

Podobně jako byly překryty SV a JV, byly nyní překryty i SZ a JZ blok. Diagram vzniklý po překrytí lze chápat dvojím způsobem:
programátorsky jako schéma provádění výpočtu a tok dat při něm, přičemž se každá hodnota určená v pravé části uchová v paměti a použije dvakrát,
ale také inženýrsky, kdy zelené čáry v diagramu představují skutečné vodiče (nebo jejich skupiny), které každý výstup z pravé části rozvádějí do dvou různých míst, kde se pak nově spárované dvojice hodnot kombinují sloupcem dvojic hradel na n výstupních hodnot. ⋄

Krok: *Opakování rekurze*

Další důležité pozorování je, že maticové bloky v SV a JV části schématu představují diskrétní Fourierovu transformaci vektorů poloviční dimenze; v horní části v bloku vzniklém překrytím původních SZ a JZ bloků se transformuje vektor $(a_0, a_2, \ldots, a_{n-2})$ a v dolní části vzniklé z SV a JV bloků vektor $(a_1, a_3, \ldots, a_{n-1})$.

K tomu stačí si uvědomit, že pokud $\omega^n = 1$, pak pro $\vartheta = \omega^2$ platí $\vartheta^{n/2} = 1$. Proto je možno $\vartheta = \omega^2$ použít namísto původního ω pro rychlou Fourierovu transformaci vektoru dimenze $n/2$. Aby se předešlo zmatku s čísly ω, ϑ a pod., používáme pro označení čísla, jehož m-tá mocnina je 1 označení ω_m. Je tedy naše původní ω číslem ω_n, ϑ číslem $\omega_{n/2}$ a platí triviální vztahy

$$\omega_n^2 = \omega_{n/2}, \quad \omega_n^n = 1, \quad \omega_{n/2}^{n/2} = 1.$$

Při dalším pokračování v úpravě přejdeme zpět na první z výše popsaných kroků a opakujeme úpravy současně v obou blocích pro výpočet FFT vektorů dimenze $n/2$ současně a cyklus bude opakován, dokud úlohu nepřevedeme na n bloků provádějících transformaci vektoru dimenze 1, která je identickým zobrazením. ◇

Scéna: *Velká dimenze*

Výhoda velké rychlosti FFT se projeví tím více, čím větší je dimenze transformovaného vektoru (neboli čím víc počítáte, tím víc ušetříte). Tato scéna je v zásadě opakováním předchozí, ale dimenze je zvýšena na $n = 64$ (větší dimenze tvaru mocniny 2 již narážejí u menších obrazovek na problémy s rozlišením).

Pro $n = 64$ se již členy ve vzorci zobrazují jako pouhé tečky; každá z nich zahrnuje jedno násobení mocninou ω a (s výjimkou levého sloupce) jeden součet. Z obrázku je vidět, že počet aritmetických operací, které mají být provedeny, je opravdu velký.

Knoflíkem **Krok** se výpočetní schéma mění tak, jak odpovídá celé jedné iteraci postupu popisovaného v předchozí scéně. Po jednom kroku se tedy převede na dvě diskrétní Fourierovy transformace vektorů dimenze 32, v dalším kroku na 4 transformace vektorů dimenze 16 atd. až nakonec se získá výsledné schéma obvodu pro výpočet FFT.

I na výsledném obrázku každá tečka představuje jedno násobení mocninou ω a jedno sčítání. Je vidět, že rozdíl mezi počtem operací, které by bylo nutno provést při výpočtu "podle definice" a počtem operací podle FFT algoritmu je obrovský. I když jsem přednášel o Fourierově transformaci po řadu let, tento obrázek pro mne byl jedním z největších překvapení při práci na Algovizi; dojem z grafického vyjádření je podstatně působivější, než pouhé konstatování,

že namísto 3969 sčítání a 4032 násobení podle definice se při výpočtu podle FFT provede pouze 384 sčítání a stejný počet násobení.

Zelené čáry ve schématu představují pro inženýra vodiče, kterými je nutno propojit výstupy jednoho sloupce hradel s vstupy následujícího sloupce hradel. Soustava vodičů mezi dvěma sloupci se dá rozdělit do tří částí: vodiče vodorovné, vodiče, které jdou ve směru JV-SZ a vodiče SV-JZ. Každá soustava je tvořena rovnoběžnými, tedy nepřekrývajícími se vodiči a proto se při výrobě integrovaného obvodu dají rozdělit do tří vrstev, z nichž každá je bez křížení, což je z technologického hlediska přijatelné.

Na obrázku vodiče zabírají větší plochu než samotná logická hradla. To není náhodné: je možno dokázat, že *jakýkoliv* obvod pro výpočet diskrétní Fourierovy transformace vyžaduje za předpokladu minimální možné vzdálenosti vodičů (což bývá konstanta vyplývající z použité technologie) zhruba stejnou plochu vodičů, jako je u zobrazovaného schématu, pokud požadujeme, aby měl obvod srovnatelnou rychlost. (Jinými slovy, obvody s menší plochou zabíranou vodiči musí být pomalé). Důkaz tohoto tvrzení nemá nic společného s výpočetní náročností, ale je založen výhradně na analýze toku dat.

Scéna: *Paralelní výpočet*

Pokud se výpočet FFT provádí na jednoprocesorovém počítači, pak pro transformaci vektoru dimenze n je nutné provést $n \log_2 n$ násobení a stejný počet sčítání (jelikož n je mocninou 2, je logaritmus celé číslo).

FFT se často používá při zpracování digitálního signálu (akustický, obrazový, atd.) v reálném čase a dimenze transformovaných vektorů může být velmi velká. V takovém případě je výhodné, ne-li přímo nutné, provádět zpracování paralelně.

Na schématu pro výpočet FFT je jasně vidět, že výpočetní jednotky v jednom sloupci operují nad různými daty a proto mohou pracovat současně. Zpracování jednoho vektoru proto vyžaduje tolik dvojic paralelních kroků, kolik je sloupců, tedy $\log_2 n$ a v každém z kroků je aktivní jeden sloupec, ve kterém se nejprve současně provádí n násobení mocninami ω a potom současně n sčítání.

Stisknutím knoflíku **Paralelní výpočet** se provede jednoduchá animace, znázorňující průchod dat obvodem při paralelním výpočtu.

Scéna: *Zřetězený výpočet*

Při zpracovávání digitálního signálu se málokdy stane, že je transformován jediný vektor. Vektory obvykle přicházejí v nepřetržitém proudu. Jak bylo vidět v předchozí scéně, při transformaci jednoho vektoru je v každém dvojkroku představovaném paralelně prováděným násobením následovaným paralelním sčítáním aktivní pouze jeden sloupec. Jakmile je jistý vektor zpracován prvním sloupcem, který je ve schématu zcela vpravo, a postoupí vlevo do dalšího

sloupce, je možno v prvním sloupci začít zpracovávat další vektor a nečekat, až první vektor bude zcela zpracován a opustí obvod. Současně je proto v obvodu možno zpracovávat tolik vektorů, kolik je sloupců, tedy logaritmický počet (vzhledem k n), přičemž vektory do obvodu vstupují s časovými rozestupy rovnými době potřebné ke zpracování vektoru jedním sloupcem výpočetních jednotek a přenosu k dalšímu sloupci. Návrh můžeme provést i jemněji tak, že každý sloupec zpracovává současně dva vektory: na jednom se provádí paralelní násobení mocninou ω a na druhém sčítání (a bylo by možné uvažovat i o zřetězeném provádění operací násobení a sčítání). Výsledkem zřetězení je tedy mimořádná výpočetní výkonnost zařízení, kde je každá výpočetní jednotka vytížena na 100 %, datový tok obvodem je přitom velmi jednoduchý a přehledný.

Stisknutím knoflíků v ovladači **Zřetězený výpočet** je možno spustit nebo krokovat jednoduchou animaci, ve které po sobe postupující vektory jsou navzájem odlišeny barvou podkladového obdélníka. Je možno zvolit přístup, kdy výpočetní jednotka je považována za nedělitelný blok, i přístup, kdy každý sloupec zpracovává současně dva vektory - jeden v násobičkách a druhý, který postupuje před ním, ve sčítačkách.

Část V

Geometrické algoritmy

Cílem této části je ukázat způsoby řešení dvou úloh, které spadají do výpočetní geometrie. I když obě úlohy je možno uvažovat pro body v rovině, třídimenzionálním prostoru nebo i prostorech vyšší dimenze, zde se omezíme na rovinu, protože v obou případech je z výpočetního hlediska velký rozdíl mezi dimenzí 2 a dimenzí větší nebo rovnou 3 a ve vícedimenzionální variantě by problémy byly příliš těžké a jejich výklad příliš obsáhlý.

První úloha je určení konvexního obalu množiny bodů v rovině. Jedná se o jednu ze základních úloh výpočetní geometrie a také o jednu z nejjednodušších úloh. Tvar řešení si na obrázku každý snadno představí a výpočetní postup, který ukážeme, je také velmi jednoduchý a rychlý. Netriviální je pouze ukázat, že počet kroků, které algoritmus provede je úměrný velikosti množiny bodů, jejíž konvexní obal konstruujeme.

Druhá úloha je naopak složitá jak na představivost, tak i výpočetně. Jedná se o nalezení Voroného diagramu množiny bodů v rovině (tyto body se tradičně označují jako *místa*). Jde o to, každému místu přiřadit množinu bodů roviny, z nichž je do daného místa blíže než do ostatních míst zadané množiny. Už úloha nakreslit ručně Voroného diagram pro 5-6 míst není vůbec jednoduchá a zrovna tak je složitý i algoritmus, který je možno použít pro řešení obecného případu. Bude popsán Fortunův algoritmus, který je založen na elegantní algoritmické myšlence; opomineme další algoritmy jako je inkrementální algoritmus, který začne s diagramem pro 3 místa a postupně přidává další a diagram příslušným způsobem upravuje nebo algoritmus typu "rozděl a panuj", který rekurzivně najde diagramy pro levou a pravou polovinu dané množiny míst a pak z nich kombinuje Voroného diagram celé množiny. Ukázány jsou také některé aplikace Voroného diagramu.

Kapitola 21

Konvexní obal bodů roviny

Množina M v rovině nebo obecněji v libovolném n-dimenzionálním Euklidovském prostoru se nazývá *konvexní*, jestliže s každými dvěma body množiny M leží v této množině i úsečka, která dané body spojuje.

Průnik *libovolného* (tedy i nekonečného) souboru konvexních množin je zjevně také konvexní. Je-li dána množina A n-dimenzionálního Euklidovského prostoru, pak průnik *všech* konvexních množin, které obsahují A je tedy *nejmenší* konvexní množina, obsahující množinu A. Tento průnik se nazývá *konvexní obal* množiny A.

V celé této kapitole se soustředíme na hledání konvexního obalu konečné množiny *v rovině*, protože určování konvexního obalu v prostorech vyšší dimenze je daleko složitější.

Scéna: *Úvod*

Tato scéna je jen ilustrací konvexního obalu množiny černě nakreslených bodů, kterým budeme ve zbytku kapitoly říkat místa. Postupte dále.

Scéna: *Zadání*

Zadáním pro výpočet je v této kapitole konečná množina bodů roviny, které budeme nazývat místa. Množinu můžeme měnit. Jednou možností je vygenerovat novou náhodnou množinu, jejíž počet prvků je dán číslem na ovladači. Místa jsou znázorněny pomocí černých koleček se středy v zobrazovaných bodech. (V některých dalších scénách bude pro některá místa používána i šedá barva).

Další možností je změna množiny myší. Při volbě **Myš neaktivní** je tato možnost vypnuta. Je-li zvoleno **Vložit místo,** pak klepnutím myší mimo stávající místa přidá další místo. Při volbě **Vynechat místo** se po klepnutí myší na existující místo toto místo vynechá. Nakonec při volbě **Pohyb Místa** je možno Místo zachytit myší a pohybovat jím.

Scéna: *Konvexní obal*

V této scéně se po klepnutí na knoflík **Obal** zadaná množina zobrazí i se svým žlutě vybarveným konvexním obalem. Knoflík **Zpět** vrátí zpět vstupní množinu bez obalu. Množinu je možno měnit, jak bylo popsáno v předchozí scéně a dívat se, jak se mění její konvexní obal. Obzvláště zábavné je táhnout některé z míst myší po celé obrazovce.

Hranice neprázdné omezené uzavřené konvexní množiny M v rovině (tedy množina bodů, které patří do M, ale bezprostředně sousedí i s body mimo množinu M) je uzavřená křivka a jedná-li se o konvexní obal konečné množiny, pak je to konvexní mnohoúhelník, jehož vrcholy patří do množiny M. Hranice konvexního obalu znázorněného na obrazovce je obtažena černou čarou.

Místa, která jsou prvky konečné množiny M je možno rozdělit do dvou částí: místa, které leží uvnitř konvexního obalu a místa, které leží na hranici konvexního obalu.

Místa uvnitř konvexního obalu zde i v dalších scénách kreslíme tmavě šedou barvou, místa na hranici zůstanou černé.

Scéna: *Orientace hranice*

Jak jsem již uvedl v minulé scéně, hranice neprázdné omezené konvexní množiny v rovině je uzavřená křivka, a proto je možno ji orientovat ve směru nebo proti směru hodinových ručiček. V této scéně bude hranice konvexního obalu obtažena zelenými šipkami s orientací ve směru hodinových ručiček.

Orientace hranice konvexního obalu konečné omezené množiny M automaticky dává i cyklické uspořádání těch míst množiny M, které leží na hranici obalu. Pro jednoznačnost je budeme psát tak, aby jako poslední (zcela vpravo) bylo uvedeno místo s největší x-ovou souřadnicí (a je-li jich více, pak ten s největší y-ovou souřadnicí).

Cyklické uspořádání míst ležících na hranici je velmi úsporný popis konvexního obalu, který přitom poskytuje přímo nebo jen na základě nenáročných úprav většinu informace, která je obvykle o konvexním obalu vyžadována. Budeme jej proto považovat za konečný výsledek výpočtu.

V této scéně jsou místa množiny očíslovaná a cyklické uspořádání dané orientací je uvedeno (vypínatelně) v spodní části obrazovky.

Zkuste si znovu pohybovat některým místem v okně nebo jinak měnit množinu, jejíž konvexní obal a cyklické uspořádání hranice je znázorněno.

Scéna: *Inkrementální algoritmus*

Algoritmus použitý k výpočtu je inkrementální. Předpokládá, že místa výchozí množiny jsou srovnány do posloupnosti $x_1, x_2, \ldots, x_n$ tak, aby jejich x-ové souřadnice představovaly neklesající posloupnost a postupně vytváří konvexní obal množiny $\{x_1, \ldots, x_k\}$ pro $k = 3, \ldots, n$ tím, že nejprve vytvoří

trojúhelník s vrcholy x_1, x_2 a x_3 s patřičnou orientací hranice a pak vždy konvexní obal množiny $\{x_1, \ldots, x_{k-1}\}$ doplní připojením místa x_k na obal množiny $\{x_1, \ldots, x_k\}$. Zkuste si výpočet krokovat v těchto hrubých krocích.

Scéna: *Postup výpočtu*

Nyní si zkuste výpočet krokovat ještě jednou. Tato scéna ukazuje detailně, jak rozšířit konvexní obal o další místo výchozí množiny.

Úsek výpočtu zahrnující rozšiřování konvexního obal o další místo u výchozí množiny budeme nazývat *fáze* výpočtu a právě přidávané místo se bude nazývat *vedoucí místo fáze*.

Fáze začíná následujícím krokem:

Krok: *Přidání nového místa*

Křivka obíhající hranici se rozšíří o nové místo tak, že z posledně přidaného místa jde do nově přidaného místa, vrací se zpět a pak již pokračuje starým způsobem. Po tomto kroku je hraniční křivka degenerovaná (z nově přidaného vedoucího místa fáze se vrací po stejné dráze, jako se do něho dostala) a množina, kterou křivka nyní ohraničuje pochopitelně není konvexní a cílem následujících kroků je ji na konvexní doplnit. ◇

Doplnění křivky z předchozího kroku na hranici konvexního obalu se provede prováděním rozšíření obalu nejprve ve směru hodinových ručiček a potom proti směru hodinových ručiček.

Rozšiřování obalu ve směru hodinových ručiček sestává z opakovaného provádění následujících dvou kroků:

Krok: *Určení úhlu ve směru hodinových ručiček*

Označme si jako u vedoucí místo fáze. Dále označme jako v a w místa křivky, které za ním následují na křivce ve směru hodinových ručiček. Místa jsou na obrazovce nakresleny barevně - u tmavě modře, v červeně a w tmavě zeleně.

Určíme úhel, o který se musí polopřímka $v \rightarrow u$ (začínající v červeném místě v a procházející modrým místem u) otočit okolo svého počátku ve směru hodinových ručiček, aby splynula s polopřímkou $v \rightarrow w$.

Namísto tohoto komplikovaného slovního výkladu se raději podívejte na animaci, popisovaný úhel je zobrazen červeně, pokud je menší než 180 stupňů a modře, pokud je větší nebo roven 180 stupňů.

Je-li úhel určený v předchozím kroku modrý, neboli větší nebo roven 180 stupňů, pak rozšiřování obalu ve směru hodinových ručiček končí a provede se skok na rozšiřování obalu proti směru hodinových ručiček. V opačném případě se pokračuje následujícím krokem elementárního rozšíření obalu. ◇

Krok: *Elementární rozšíření obalu ve směru hodinových ručiček*

Trojúhelník určený místy u, v, w přidá k vytvářenému konvexnímu obalu a současně s tím místo v přestane ležet na hraniční křivce na úseku z vrcholu u do vrcholu w. Poté se skočí zpět na krok určování úhlu ve směru hodinových ručiček. ◇

Po ukončení kroků ve směru hodinových ručiček se provádějí obdobné kroky proti ve směru hodinových ručiček

Krok: *Určení úhlu proti směru hodinových ručiček*

Označme si jako u vedoucí místo fáze. Dále označme jako v a w místa, která potkáme, jdeme-li po křivce z místa u proti směru hodinových ručiček. Místa jsou na obrazovce nakresleny barevně - u tmavě modře, v červeně a w tmavě zeleně.

Určíme úhel, o který se musí polopřímka $v \to u$ (začínající ve w a procházející místem u) otočit okolo svého počátku proti směru hodinových ručiček, aby splynula s polopřímkou $v \to w$.

V animaci je zase popisovaný úhel je zobrazen červeně, pokud je menší než 180 stupňů a modře, pokud je větší nebo roven 180 stupňů.

Je-li úhel modrý, fáze končí a žlutá množina je konvexním obalem dosud zpracovaných míst. Pokud je úhel červený, pokračuje se následujícím elementárním rozšířením obalu. ◇

Krok: *Elementární rozšíření obalu proti směru hodinových ručiček*

Trojúhelník určený místy u, v, w přidá k vytvářenému konvexnímu obalu a současně s tím místo v přestane ležet na hraniční křivce. Poté se skočí zpět na krok určování úhlu proti směru hodinových ručiček. ◇

V dalším budeme červenému vrcholu v z kroku určování úhlu říkat pivot kroku.

Scéna: *Datová struktura*

Jak bylo již uvedeno, pro popis konvexního obalu množiny používáme posloupnost míst množiny ležících na hranici jejího konvexního obalu a to v cyklickém uspořádání, které dostaneme při oběhu hranice ve směru hodinových ručiček. Zápis je volen tak, že končí nejvýchodnějším místem konvexního obalu a pokračuje ve směru hodinových ručiček dalšími místy množiny v cyklickém uspořádání.

Tato scéna umožňuje krokovat určování konvexního obalu v hrubých inkrementačních krocích i detailně a přitom na spodní straně obrazovky je napsána posloupnost míst zmíněná v předchozím odstavci. Místa množiny jsou očíslovány pro identifikaci v hraniční posloupnosti. Barva pozadí prvků cyklické posloupnosti sleduje při určování úhlů změnu barvy odpovídajícího místa množiny, jejíž obal hledáme.

Je vidět, že operace přidání nového místa spočívá v překopírování dosavadního posledního prvku posloupnosti také na začátek a pak přidání nového místa na konec posloupnosti. Operace elementárního rozšíření obalu spočívá ve vynechání prvního prvku posloupnosti (při postupu ve směru hodinových ručiček) nebo předposledního prvku (při postupu proti směru hodinových ručiček).

Zkuste si výpočet krokovat a sledovat změny posloupnosti. Před započetím výpočtu nebo po jeho dokončení je možno množinu i měnit a sledovat jak se v návaznosti na to mění popis konvexního obalu.

Scéna: *Složité přidání místa*

Nakonec se budeme zabývat výpočetní složitostí algoritmu (předpokládajíce, že místa jsou setříděna podle x-ové souřadnice). Úvodní trojúhelník se zkonstruuje snadno v konstantním čase a pak následuje $n-3$ inkrementačních fází, kdy v každé rozšíříme konvexní obal o jedno místo z výchozí množiny. Nyní jde o to, jak dlouho trvá jedna inkrementační fáze, tedy o to, kolikrát v jedné fázi provádíme určování úhlu sevřeného polopřímkami a případné zvětšení oblasti.

V této scéně začínáme výpočet "z prostředka", přesněji řečeno jsme na počátku poslední fáze. Je již určen konvexní obal všech míst s výjimkou jednoho, toho který se nachází na pravé straně obrazovky, a v této fázi dosud sestrojený obal budeme rozšiřovat na obal celé množiny.

Z této konfigurace si krokujte výpočet. Uvidíte, že v tomto případě počet kroků v jedné fázi je takřka rovný počtu míst množiny. Poslední fáze tedy pro danou množinu míst trvá velmi dlouho.

Zkuste se v předváděném případě vrátit na začátek a uvidíte, že předchozí fáze byly velmi krátké. To není náhoda; jak uvidíme v následující scéně, určení úhlů se v celém výpočtu provede nejvýše $4n$, kde n je velikost množiny, ale (jak jsme právě viděli) délka fází může silně kolísat.

Dobrý časový odhad tedy nedostaneme tak, že bychom maximální časový odhad pro jednu fázi vynásobili počtem fází. Jak nejlépe odhadnout počet kroků algoritmu uvidíme v následující scéně.

Scéna: *Výpočetní složitost algoritmu*

Je jasné, že počet fází výpočtu je roven velikosti vstupní množiny.

Operace přidání nového místa (viz scéna Postup výpočtu) se v každé fázi provede jen jednou a trvá konstantní čas. Celkově tedy přidávání nového místa zabere během celého výpočtu čas úměrný velikosti vstupní množiny.

Algoritmus dále opakuje kroky určení úhlu a případného elementárního rozšíření obalu. Jak bylo ukázáno v předchozí scéně, počet opakování tohoto dvojkroku může být v jedné fázi velký a proto určíme počet opakování těchto dvojkroků za celý výpočet dohromady.

Použijeme k tomu jako v řadě jiných algoritmů vhodného účetnictví. Každý vrchol bude mít svůj účet, na začátku výpočtu nulový. Po provedení každého dvojkroku bude přičtena jedna jednotka na účet některého vrcholu a nakonec počet provedených dvojkroků dostaneme tak, že sečteme účty všech vrcholů.

Dokážeme, že následující způsob účtování způsobí, že na konci výpočtu nebude mít žádný vrchol na účtu více než 4:

- určení úhlu, kdy určený úhel je větší nebo rovný 180 stupňů (modrá výseč - tedy bez následného elementárního rozšíření obalu), se přičítá na účet vedoucího místa fáze (nejvíce vpravo z uvažovaných míst - tmavě modré místo u);

- určení úhlu, kdy určený úhel je menší než 180 stupňů (červená výseč - tedy s následným elementárním rozšířením obalu), se přičítá na účet pivotu kroku (červený vrchol v).

V každé fázi se modrá výseč objeví jen dvakrát, protože její první výskyt končí postup ve směru hodinových ručiček a její druhý výskyt končí postup proti směru. Každý vrchol je vedoucím vrcholem fáze jen v jedné fázi a proto každý vrchol bude na konci výpočtu mít na svém účtu jen dvě jednotky za modré výseče.

Jestliže se vrcholu přičte jedna jednotka za červenou výseč jako pivotu kroku postupujícího ve směru hodinových ručiček, pak se tento vrchol vzdálí od spodní části hraniční křivky během přidání trojúhelníka určeného vedoucím vrcholem fáze, uvedeným pivotem a následujícím (tmavě zeleným) vrcholem k budovanému konvexnímu obalu. Takový vrchol se ale již nikdy více nemůže stát pivotem v kroku určování úhlu ve směru hodinových ručiček. Z toho plyne, že každý vrchol může mít na svém účtu jen jednu jednotku za roli pivota v kroku prováděném ve směru hodinových ručiček.

Podobně jestliže se vrchol stane pivotem kroku prováděného protisměrně, vzdálí se od horní části hraniční křivky a proto se již nikdy nemůže stát pivotem v žádném následujícím kroku prováděném proti směru hodinových ručiček. Za roli pivota v protisměrném kroku může tedy každý vrchol dostat také nejvýše jednu jednotku.

Tím je dokázáno, že počet dvojkroků bude za celou dobu výpočtu nejvýše $4n$. Jak jsme viděli, některé fáze mohou trvat velmi dlouho, ale z toho, co bylo řečeno, plyne, že je to vykompenzováno tím, že jiné fáze budou zase kratší než průměrný počet 4 dvojkroky.

V této scéně jsou u jednotlivých míst nakresleny jejich účty; sledujte podrobně, jak se stavy účtů mění v souladu s výše provedenými úvahami.

Je také možno přepnout do módu, kdy se stav účtu nezobrazuje numericky, ale malými červenými kolečky (mincemi), které jsou modré nebo červené podle

toho, zda byly přidány na účet v kroku, ve kterém byla určená úhlová výseč modrá nebo červená. Pak je dobře vidět, že místo dostane na svůj účet dvě modré mince v době, kdy je vedoucím místem právě probíhající fáze, a pak nejvýše dvě červené mince v krocích, ve kterých je pivotem.

Odhad by bylo možno ještě trochu vylepšit: místa, která dostanou na svůj účet dvě červené mince, se vzdálí od horní i dolní části hranice obalu a dostanou se tedy dovnitř konvexního obalu zpracovávané množiny. Místa, která budou na konci na hranici konvexního obalu proto budou mít na svém účtu nejvýše 3 mince - nejvýše dvě modré a nejvýše jednu červenou. Počet provedených dvojkroků tedy bude celkem za celý výpočet nejvýše $4n$ minus počet míst, které zůstanou na hranici obalu.

Aby se přidávání mincí na účty lépe sledovalo, je vždy posledně přidaná mince (tedy mince přidaná za právě prováděnou akci) graficky zvýrazněna. Toto zvýrazňování ale je také možno na ovladači vypnout.

Kapitola 22

Voroného diagram bodů roviny

Scéna: *Místa*

Na obrazovce jsou černými kolečky znázorněny body v rovině, které budeme nazývat *místa*. Množina míst představuje vstupní data pro hledání Voroného diagramu. V této scéně si jenom ukážeme, jak se vstupní množinou míst manipulovat.

Zde i ve většině následujících scén je možno vytvořit novou množinu míst knoflíkem **Nový vstup,** její velikost se může nastavit v poli vedle návěští **N=.** Ve scénách, kde jsou přístupné, mohou být použity knoflíky **Zvětši** a **Zmenši** pro zoomování obrazu, které mnohdy přinese zajímavý nový pohled na celkovou konfiguraci.

Na ovladači je také volba funkce myši (použitá i v některých dalších scénách)
Volba **Pohyb diagramem:** Myší je možno posouvat rovinu, ve které se nacházejí místa a Voroného diagram.
Volba **Pohyb místa:** Myší je možno zachytnout místo a táhnout jej libovolně v okně (výpočetně náročná operace, počítač často nestíhá).
Volba **Přidání místa:** Klepnutí myší do prázdné části obrazovky se vytvoří nové místo.
Volba **Vynechání místa:** Klepnutím myší na existující místo se toto vymaže.

Scéna: *Voroného diagram*

Tato scéna ukazuje Voroného diagram pro zvolenou množinu míst. Kolem každého místa se nachází oblast ve tvaru konvexního mnohoúhelníka, ohrani-

čeného černými čarami. Některé z hraničních čar jsou úsečky konečné délky, jiné jsou polopřímky.

Výjimečně se může stát, že hranicí je přímka (pak je oblastí polorovina) nebo 2 rovnoběžné přímky (pak je oblastí nekonečný pás), ale je-li míst více než 2, je to mimořádně nepravděpodobné. Tyto situace budou ukázány ve scéně zabývající se degenerovanými případy.

Body roviny se rozdělí do několika kategorií:

- Vnitřek oblasti kolem místa s je tvořen těmi body roviny, ze kterých je do místa s blíže než do libovolného jiného místa z uvažované množiny.

- Body, které leží na hranici oblastí příslušných místům s_1 a s_2, jsou většinou ty body, ze kterých je do s_1 a s_2 stejně daleko a přitom blíže než do jakéhokoli jiného místa. Takové body leží na dělících úsečkách, polopřímkách nebo přímkách, ale nejsou jejich koncovými body.

- Dále jsou v diagramu body, které leží současně na hranici oblastí tří různých míst s_1, s_2 a s_3. Tyto body jsou stejně vzdáleny od všech tří míst s_1, s_2 a s_3 a přitom jsou jim blíže než libovolnému dalšímu místu. Jedná se o koncové body dělicích úseček a polopřímek.

- Obecně se mohou vyskytnout i body x, které mají čtyři nebo více nejbližších míst. Tento případ je ale u náhodně zvolené množiny míst velmi nepravděpodobný: bod x by byl na hranici alespoň čtyř oblastí (jako čtyřbod Four Corners na hranici Arizony, Colorada, Nového Mexika a Utahu v USA) a jeho nejbližší místa by musela všechna ležet na kružnici se středem v x. Takovéto a další málo časté degenerované případy rozebereme podrobněji dále.

Body, které jsou na hranicích tří nebo více oblastí se nazývají *Voroného body*.

Měňte soubor míst jejich posunem, přidáváním a vynecháváním a pozorujte, jak se diagram mění.

Scéna: *Fortunův algoritmus*

Tento applet ukazuje, jak se Voroného diagram nalezne algoritmem, který popsal S. Fortune. Algoritmus bude také v příštích dvou scénách vykládán v 3D nákresu. Abych předešel možným nedorozuměním týkajících se směru "nahoru" a "dolů", budu horní část obrazovky zásadně označovat jako "severní" a dolní část obrazovky jako "jižní", užívajíce konvenci běžnou v kartografii. Pojem "horní" a "dolní" budu používat jen v 3D representaci, kdy předpokládám, že se na situaci díváme shora a proto "horní" bude znamenat "blíže k pozorovateli" a "dolní" bude "dále od pozorovatele."

Fortunův algoritmus je typu “sweep line” (zametací přímka). V této scéně se po chvíli objeví na obrazovce přímka, která “zametá” rovinu od jihu k severu a v každém okamžiku ukazuje Voroného diagram pro body, které už byly zametací přímkou dosaženy - přesněji řečeno tu část tohoto diagramu, která se již nebude měnit a zůstane ve své podobě jako součást výsledného diagramu celé množiny míst. Zametací přímka není ve skutečnosti explicitně nakreslena, ale je dána pohybujícím se rozhraním mezi tmavou a světlou částí roviny.

Běh algoritmu je možno zastavit knoflíkem **Stůj a** znovu spustit knoflíkem **Vpřed a** nebo jej pustit pozpátku knoflíkem **Vzad.** Je také možné skočit na konec výpočtu knoflíkem **Konec** nebo se vrátit na začátek knoflíkem **Začátek** a knoflíkem **Vpřed** nebo **Vzad** znova algoritmus spustit. Zametací přímkou je také možno pohybovat myší: při zmačknutí myši mimo ovladač zametací přímka skočí tak, aby procházela bodem, kde bylo myší klepnuto a při pohybu myši při stisknutém knoflíku myši se bodu určeného kurzorem drží.

V jisté vzdálenosti na jih od zametací přímky se objevuje červeně nakreslená čára, která se pro svůj tvar v anglické literatuře označuje jako “beachline” - pobřežní čára. Jak je vidět, Voroného diagram je kreslen jen v oblasti *na jih* od pobřežní čáry. Je to proto, že (jak ukážeme dále) tvar výsledného diagramu v oblasti na jih od pobřežní čáry je jednoznačně dán místy, které už byly “zameteny” zametací přímkou, zatímco v oblasti mezi pobřežní čárou a zametací přímkou může být ovlivněn místy, které ještě zametací přímkou dosaženy nebyly. Pochopitelně v oblasti na sever od zametací přímky ještě také není možno určit, jak výsledný diagram bude vypadat.

Při podrobnějším zkoumání zjistíme, že pobřežní čára je složena z řady oblouků (jak dále uvidíme, jsou to části parabol) a body, ve kterých se tyto oblouky setkávají, kreslí Voroného diagram. Proč tomu tak je uvidíme v následujících scénách.

Scéna: *Kužely*

Tato scéna přináší alternativní pohled na situaci, která byla popsána v předchozích scénách. Přináší pohled, který může velmi usnadnit chápání algoritmické myšlenky Fortunova algoritmu, ale některým čtenářům naopak působí obtíže, protože vyžaduje prostorovou představivost a schopnost si vytvořit třídimenzionální obrázek na základě dvoudimenzionálního náčrtu. Jelikož není bezpodmínečně nutná pro výklad algoritmu, je možno ji přeskočit a po případě se k ní vrátit později, i když si myslím, že pochopení algoritmu usnadňuje a prohlubuje.

Zobrazení této scény je výpočetně náročné, může se proto stát, že při jejím kreslení a překreslování bude docházet k prodlevám.

Scéna znázorňuje stejnou situaci jako scény předcházející, ale znázorněnou nikoli v rovině, nýbrž v třídimenzionálním prostoru. Rovina, ve které se na-

chází místa, jejichž Voroného diagram hledáme, je vnořena do 3D prostoru. Můžeme si představovat, že tato rovina je vodorovná a díváme se na ní shora. Předpokládáme, že se díváme z takové dálky, že jevy související s perspektivou byly potlačeny a na obrazovce je rovnoběžný vertikální průmět do roviny obsahující místa.

Z každého místa je spuštěn rotační kužel, který je vystínován jakoby na něj dopadalo zleva světlo. Spádnice kuželů mají úhel 45 stupňů. Celá situace tedy znázorňuje pohoří, ve kterém hory mají pravidelný a shodný tvar a jejich vrcholy (naše místa) jsou ve stejné výšce. Při pohledu shora pochopitelně kužely nevidíme celé, neboť se navzájem překrývají; z každého kuželu vidíme tu část, která shora není zakryta jiným kuželem.

Pohybem myši nahoru a dolu se stisknutým levým knoflíkem je možno pohoří naklápět - naklopte jej tak, abyste se na něj dívali z boku. Pohybem myši zleva doprava se pohořím otáčí. (Pohoří není zobrazeno celé, ale jen jeho výřez, který byl v okně vidět při vertikálním pohledu; při testování většina uživatelů tento způsob považovala za lépe srozumitelný). Knoflíkem **Vertikální** je možno vše vrátit do výchozí polohy.

Klepněte nyní pravým knoflíkem myši na libovolný bod na kuželové ploše. Objeví se pravoúhlý rovnoramenný trojúhelník, který má každou stranu jiné barvy (pokud vidíte jen červenou čáru nebo je trojúhelník velmi úzký, natočte si pohoří tak, aby byl lépe vidět). Zeleně nakreslená přepona trojúhelníka ukazuje sestup z vrcholu kužele, na jehož viditelné části zvolený bod sedí, po spádové přímce kužele do zvoleného bodu. Červená odvěsna je průmět zeleného sestupu do vodorovné roviny, ve které leží vrcholy kuželů (neboli výchozí místa). Nakonec žlutá odvěsna ukazuje, jak hluboko je zvolený bod pod rovinou vrcholů kuželů. Trojúhelník je pravoúhlý, protože žlutá odvěsna je kolmá na rovinu míst a tedy i na červenou odvěsnu, a jelikož spádové přímky kuželů klesají pod 45 stupni, je také rovnoramenný.

Pohybujte kurzorem se stisknutým pravým knoflíkem myši, abyste trojúhelník viděli v různých polohách, popřípadě si (se stisknutým levým knoflíkem myši) můžete pohoří vhodně naklopit, aby trojúhelník byl dobře viditelný. Nakonec převeďte pohoří do základní polohy (například knoflíkem **Vertikální)** abyste si ověřili, že to co v této poloze vidíte jako spojnici vrcholu kužele a zvoleného bodu je ve skutečnosti průmět jejich skutečné spojnice v prostoru.

Asi jste si všimli, že pokud se při pohybu zvoleným bodem dostanete tam, kde se protínají viditelné části dvou kuželů, objeví se trojúhelníky dva (a ve Voroného bodech se mohou objevit i tři). Nastavte zvolený bod tak, aby ležel na průsečíku dvou kuželových ploch a natočte pohoří tak, abyste mohli trojúhelníky dobře pozorovat. Oba trojúhelníky jsou pravoúhlé a rovnostranné a navíc mají společnou žlutou odvěsnu. Z toho okamžitě plyne, že jsou shodné a tedy mají také stejně dlouhé zelené přepony a pro nás je důležité, že mají stejně dlouhé i červené odvěsny. Přetočíte-li nyní pohoří do základní polohy,

ve které je sledujeme svisle shora, pak délky shodných červených odvěsen udávají vzdálenosti ze zvoleného bodu do vrcholů kuželů, jak se jeví v rovinném průmětu. Tím je dokázáno to, co vám už je jistě delší dobu jasné: promítneme-li si průsečíky viditelných částí kuželů do vodorovné roviny obsahující vrcholy kuželů, dostaneme Voroného diagram množiny výchozích míst (vrcholů kuželů).

Pokud nechcete, aby byl zobrazován trojúhelník (nebo trojúhelníky), zrušte na ovladači zaškrtnutí **Trojúhelník.** Vrátíte-li funkci **Trojúhelník** zpět, je třeba někam klepnout pravým knoflíkem myši, aby se trojúhelník znovu objevil.

Scéna: *Zametací rovina*

Představme si nyní, že pohoří z minulé scény je "zametáno", ale nikoliv přímkou, nýbrž rovinou, která je skloněna pod úhlem 45 stupňů (z roviny je zde nakreslen jen obdélníkový výřez). Hory (nebo jejich části), které jsou pod rovinou, jsou znázorněny různými odstíny modré, zatímco části, které se za chvíli objeví nad rovinou, budou zbarveny do šeda (od světlé po tmavou). Průsečík roviny a povrchu kuželů bude zvýrazněn červenou barvou.

Stiskněte knoflík **Vpřed** a sledujte pohyb roviny. Rovina postupně odhaluje části kuželů. Počkejte až bude zhruba v polovině pohoří a stiskněte knoflík **Stůj.** Poté, co si obrázek pozorně prohlédnete a popřípadě podle libosti natočíte, stiskněte knoflík **Vertikální.** Pohoří protnuté rovinou se natočí do polohy, kterou známe z minulé scény, kdy se na kužely díváme vertikálně shora ve směru rovnoběžném s jejich osami.

Po natočení průsečík zametací roviny a vodorovné roviny míst bude představovat zametací přímku z předchozích 2D scén. Jistě si na první pohled všimnete, že průsečík zametací roviny a povrchů kuželů (nebo přesněji jeho průmět do roviny míst - vrcholů kuželů - tedy tak, jak je nakreslen na obrazovce) je pobřežní čára, která byla vidět v rovinné animaci algoritmu. Body roviny, které jsou na jih od pobřežní čáry z předchozích 2D scén jsou vlastně průměty těch bodů na povrchu pohoří, které jsou *nad* zametací rovinou v 3D scéně, body roviny severněji od pobřežní čáry z předchozích 2D scén (ať jsou v jakékoli poloze k zametací přímce), jsou body na povrchu pohoří, které jsou *pod* zametací rovinou v 3D scéně.

První okamžik, kdy zametací rovina narazí na kužel nějakého místa je okamžik, kdy narazí na vrchol kužele a zároveň se kužele dotýká ve spádnici, která z vrcholu běží přímo k jihu.

Nyní je také jasné, jak je pobřežní křivka konstruována: jako průmět průsečíku pohoří a zametací roviny se skládá z průmětů viditelných částí průsečíků jednotlivých kuželů a zametací roviny. Jelikož je zametací rovina rovnoběžná se spádovou přímkou kužele, jsou tyto průsečíky paraboly a snadno se zjistí, že jejich průměty do vodorovné roviny jsou také paraboly. Pobřežní přímka v

2D scénách je tedy skutečně složena z parabolických oblouků.

Údolí pohoří kuželů jsou tvořena body, jejichž průměty vytvoří hranice oblastí Voroného diagramu v 2D. Tyto body jsou také ty, kde se stýkají viditelné části průsečíků zametací roviny s kužely, ohraničujícími údolí. Tím je tedy dáno, proč body v 2D reprezentaci, ve kterých se stýkají oblouky pobřežní čáry, "kreslí" hranice Voroného oblastí: jsou to přesně průměty bodů údolí.

Scéna: *Paraboly jednotlivě*

V předchozí scéně jsme objasnili význam pobřežní čáry pomocí 3D konfigurace. V této a následujících scénách vyložíme význam pobřežní čáry ještě jednou, ale na základě rovinného nákresu. I zde se ale vyskytnou kuželosečky, avšak nikoli jako průsečík kužele a roviny, ale jako množina bodů stejně vzdálených od bodu a přímky.

Jak jsem již řekl výše, algoritmus se snaží zkonstruovat tu část Voroného diagramu, kterou lze určit na základě znalosti míst, které již přešla (nebo se jich alespoň dotkla) zametací přímka. Takovým místům budeme říkat *známá místa*. Místa, která leží v tmavé severní oblasti, kterou zametací přímka ještě neprošla, budou *neznámá místa*.

Na obrázku jsou znázorněna všechna místa a zametací přímka. Klepněte na jedno známé místo. To se tím zvýrazní a objeví se růžová oblast, shora ohraničená parabolou. Hraniční parabola je množina všech bodů, které jsou stejně daleko od zvoleného místa a od zametací přímky (v okamžité poloze). Je známo, že taková křivka je parabola, která má *ohnisko* ve zvoleném místě; zametací přímka je *řídící přímka* této paraboly. Parabola je otevřena směrem k jihu a body růžové oblasti jsou ty body, ze kterých je blíže do ohniska paraboly než k její řídící přímce, zatímco body, které nejsou růžové a neleží na parabole jsou (bez ohledu na jejich polohu vůči zametací přímce) blíže k řídící přímce paraboly než k jejímu ohnisku.

Povšimněte si, že na sever od ohniska prochází parabola pochopitelně bodem, který je v polovině mezi ohniskem a řídící přímkou.

Zvolte myší libovolné jiné místo, opětně se zobrazí parabola a růžová oblast bodů bližších ohnisku než řídící přímce. Řídící přímkou lze pohybovat pomocí knoflíků jako v minulých scénách. Pohyb se zastaví, pokud by se zvolené místo dostalo na sever od zametací přímky (mezi neznámá místa). Pohyb zametací přímky myší je blokován, aby nebyl v kolizi s volbou míst myší.

Je dobré, abyste sledovali změnu tvaru paraboly při pohybu zametací přímky, tvořící řídící přímku paraboly. Pokud se řídící přímka vzdaluje od ohniska směrem k severu, parabola se pohybuje za ní a výrazně se rozevírá, naopak pokud je řídící přímka blízko ohniska, parabola je úzká.

Když ohnisko (tedy zvolené místo) leží na řídící přímce, množina bodů, které jsou stejně vzdáleny od ohniska a od řídící přímky, není parabola, ale

přímka procházející ohniskem a kolmá na řídící přímku. I tento případ bude při výpočtu podle Fortunova algoritmu důležitý a nastává při t.zv. místní události. Nás z uvedené přímky bude zajímat pouze její část (polopřímka) z ohniska na jih. Výše uvedená růžová oblast je nyní prázdná; když ohnisko leží na řídící přímce, žádný bod nemůže být blíže k ohnisku než k přímce.

Poznamenejme, že vše v této scéně by bylo možno zobrazit i v 3D provedení. Degenerovaná situace popsaná v předchozím odstavci by pak spočívala v tom, že zametací rovina zahrnuje vrchol kužele a dotýká se jeho povrchu v jediné přímce, zatímco v základní situaci zametací rovina kužel prosekává.

Scéna: *Paraboly společně*

V této scéně jsou nakresleny současně všechny paraboly známých míst, které byly popisovány v předchozí scéně. Růžově je vykreslena oblast zahrnující body, které jsou v růžové oblasti pro alespoň jedno známé místo. Jsou to přesně ty body, ze kterých je do některého známého místa blíže než k zametací přímce.

Červeně jsou nakresleny body, které jsou na parabole alespoň jednoho známého místa, ale pro žádné známé místo nepatří do jeho růžové oblasti. Jsou to body, které mají do nejbližšího známého místa stejně daleko jako k zametací přímce. Je jasně vidět, že tato množina je pobřežní čára, o které jsme mluvili v předchozích scénách.

Ostatní body jsou pak pochopitelně ty body, ze kterých je k zametací přímce blíže, než k nejbližšímu známému místu, a to bez ohledu nad to, v jakém vztahu jsou k zametací přímce. Jsou to tedy body na sever od pobřežní čáry.

Scéna: *Události*

Tvar pobřežní čáry se při jejím pohybu k severu neustále mění, ale počet oblouků a jejich rozložení se mění jen málokdy. Okamžiky, kdy se se pobřežní čára mění radikálním způsobem, nazýváme *události*. Tato scéna umožňuje animovat posun pobřežní čáry knoflíky **Nahoru** a **Dolů,** ale pohyb se automaticky zastaví nejen pomocí knoflíku **Zastavit,** ale i v okamžiku, kdy dojde k události.

Zkuste si pohyb zametací přímky k severu a sledujte přitom, při jakých událostech se pohyb zastaví. Uvidíte, že nastává jeden ze dvou případů:

- **Místní událost:** Zametací přímka právě narazila na další místo (odtud název). V pobřežní čáře se objevil se nový "oblouk", ve skutečnosti úsečka, která je částí polopřímky, představující degenerovanou parabolu nového známého místa, ležícího zatím na zametací přímce. Tím došlo k rozetnutí oblouku ležícího pod nově objeveným místem na dvě části.

- Kružnicová událost: Jeden z oblouků pobřežní čáry byl "utlačován" dvěma sousedními oblouky, jeho šířka se postupně zmenšovala a právě v tomto okamžiku zcela zmizel. Místo, kde oblouk zmizel, je označeno zeleným kolečkem. (Název události bude objasněn později.)

Velmi vzácně může docházet i k některým jiným událostem, které ale lze chápat jako současný výskyt dvou nebo více jednoduchých událostí, jako je třeba současné vymizení dvou sousedních oblouků v týž okamžik a v témže bodě roviny nebo místní událost, při které polopřímka spuštěná z nově nalezeného místa protne pobřežní čáru nikoli uvnitř oblouku, ale v bodě styku dvou oblouků, ale k jejich výskytu je třeba, aby množina míst zahrnovala některé speciální a málo časté konfigurace, ke kterým u náhodně zvolených míst v obecné poloze dochází jen zřídka, například čtyři nebo více bodů, ležících na téže kružnici. Tyto konfigurace probereme podrobněji na konci kapitoly.

Pohrajte si se scénou a sledujte situace, kdy dochází k událostem. Doporučuji ještě myší mírně posouvat v okolí polohy, kde k události došlo, pro prozkoumání konfigurace těsně před nebo těsně po události.

Scéna: *Místní událost*

Scéna přináší stejnou situaci i možnosti ovládání jako předchozí scéna, avšak pohyb zametací přímky se automaticky zastavuje pouze při místní události.

Prozkoumejme situaci v okamžiku místní události a těsně před ní a po ní trochu blíže. Těsně před místní událostí nic nenaznačuje, že by k ní mohlo dojít. Situace na sever od zametací přímky je neznámá a neexistuje žádná možnost jak zjistit, že se přímka blíží k neznámému místu.

V okamžiku, kdy došlo k události se na zametací přímce objevilo dosud neznámé místo. Do pobřežní čáry přibyla degenerovaná parabola tohoto místa. Předpokládajíce nedegenerovanou místní událost, tato polopřímka-parabola rozetnula některý z dosavadních oblouků pobřežní čáry na dvě části a vklínila se mezi ně. Průsečík je bod, ze kterého je stejně daleko do nového místa i ohniska rozetnutého oblouku a proto v něm začíná vznikat nový segment Voroného diagramu, zatím představovaný jediným bodem, průsečíkem.

Posunujte nyní pomocí myši zametací přímku velmi pomalu směrem k severu. Z degenerované paraboly se stala skutečná parabola, která je nejprve velmi úzká, ale prudce se rozšiřuje. Průsečíky větví této paraboly se západní a východní částí rozetnutého oblouku se rychle vzdalují a kreslí ze dvou stran nově vznikající segment Voroného diagramu. Směry pohybu průsečíků jsou navzájem opačné, protože se oba pohybují po ose úsečky spojující nové místo s ohniskem rozetnutého oblouku, na které musí ležet body stejně vzdálené od těchto dvou míst.

Z technického hlediska bývá při programování výhodné představovat si, že nově vznikající segment není kreslen obousměrně, ale že se ve skutečnosti

jedná o dva segmenty, které oba začínají v bodě, kde degenerovaná parabola rozetnula oblouk pod ní. Tyto dva segmenty jsou pak jednosměrně kresleny dvěma průsečíky větví paraboly nového místa se západní a východní části rozetnutého oblouku. Zatrhněte checkbox **Polopřímky** pro ilustraci tohoto přístupu, kdy oba segmenty jsou kresleny jako šipky s explicitně vyznačeným počátečním bodem.

Scéna: *Kružnicová událost*

Zhruba řečeno je kružnicová událost zánik některého oblouku pobřežní čáry, při kterém se dotknou oblouky, které s ním po obou stranách sousedily. Obrazně je možno říci, že oblouk zanikl v důsledku expanze svých sousedů.

Na rozdíl od místní události se kružnicová událost dá předpovědět dopředu, pomineme-li fakt, že ještě předtím, než k ní dojde, může nastat jiná událost, která celou situaci změní natolik, že k předvídané kružnicové události nakonec nedojde.

Nechte zametací přímku postupovat (knoflík **Vpřed** nebo **Vzad)** až se zastaví v okamžiku, kdy došlo ke kružnicové události a potom ji vraťte o něco zpět k začátku výpočtu.

Situaci nyní vidíme krátce předtím, než dojde ke kružnicové události během níž má vymizet jistý kratičký oblouk α s ohniskem s_α tím, že jej vytlačí jeho sousední oblouky β a γ s ohnisky s_β a s_γ. Přímky, které jsou osy úseček s_betas_α a $s_\alpha - s_\gamma$, po kterých se pohybují styčné body dvojic oblouků (β, α) respektive (α, γ), se protínají v bodě, do kterého pobřežní čára zanedlouho dorazí. Do průsečíku přímek tedy směřují krajní body oblouku α, v něm se setkají a v něm oblouk α zmizí.

Nechte přímku znovu doběhnout k uvažované kružnicové události a zatrhněte checkbox **Paraboly.** Objeví se úplně vykreslené paraboly míst s_β a s_γ, které nesou oblouky β a γ, jež zahubily oblouk α. Průsečíkem, ve kterém oblouk α úplně zmizel, ale také ještě prochází parabola s ohniskem s_α (je nakreslena také, ale zeleně), protože oblouk α je vlastně stále ještě součástí pobřežní čáry, i když zdegeneroval do jediného bodu. Bod c, kde oblouk α zmizel, tedy leží najednou na třech parabolách s ohnisky α, β a γ.

Z uvedeného plyne, že vzdálenosti z průsečíku c do všech tří míst s_α, s_β a s_γ jsou stejné jako vzdálenost z c k zametací přímce. Z toho plyne, že kružnice se středem v c a procházející místem s_α prochází také místy s_β a s_γ a zametací přímka se této kružnice v okamžiku vymizení oblouku α shora dotýká. Pro zobrazení této kružnice zatrhněte checkbox **Jedna kružnice.**

Teď již tedy je jasné, jak dopředu určit zda, kde a kdy zanikne oblouk α se sousedními oblouky β a γ (pokud nedojde k interferenci s jinou dřívější událostí):

1. proložíme body s_α, s_β a s_γ kružnici; zánik oblouku α lze očekávat, pokud střed této kružnice leží v oblasti, kam pobřežní čára teprve dorazí;

2. pokud k zániku oblouku α dojde, stane se tak ve středu uvedené kružnice;

3. pokud k zániku oblouku α dojde, stane se tak v okamžiku, kdy se bude zametací přímka shora dotýkat sestrojené kružnice, neboli když y-ová souřadnice bodů zametací přímky bude o r větší než y-ová souřadnice středu kružnice, kde r je poloměr kružnice.

Role kružnice při předvídání kružnicové události objasňuje název tohoto typu události.

Scéna: *Kružnicová událost - plánování*

Zkuste si nyní při zastaveném výpočtu zvolit **Jedna kružnice** a kliknout některý z oblouků pobřežní čáry. Pokud se dá předpokládat, že zvolený oblouk bude v budoucnu zahuben svými sousedy, objeví se kružnice, která prochází ohnisky zvoleného oblouku a jeho sousedů a kterou je možno chápat jako předpověď možné budoucí kružnicové události: zvolený oblouk by měl zaniknout v bodě, který je středem kružnice a to v okamžiku, kdy se bude zametací přímka kružnice dotýkat shora.

Pokud naopak se zvolený oblouk při postupu zametací přímky k severu bude rozpínat (jeho koncové body se budou vzdalovat), žádná kružnice se neobjeví.

Zvolíte-li volbu **Všechny kružnice,** zobrazí se kružnice pro všechny předpokládané budoucí kružnicové události najednou. Obrázek je obvykle trochu nepřehledný, ale ukazuje, kolik kružnicových událostí je třeba naplánovat.

Scéna: *Neuskutečněná kružnicová událost*

Tato scéna ukazuje několik modelových konfigurací, ve kterých se předpokládalo, že dojde ke kružnicové události, při které zvýrazněný oblouk pobřežní čáry zmizí, ale potom k naplánované události nedojde. Kružnice odpovídající neuskutečněné naplánované události je znázorněna také.

V příkladu 1 je zvýrazněný střední oblouk rozseknut parabolou místa, které bylo objeveno v okamžiku místní události, ke které došlo před plánovanou kružnicovou událostí.

V příkladech 2 a 3 je jeden ze sousedů zvýrazněného středního oblouku eliminován místní událostí, ke které došlo dříve. Nicméně v tomto případě je možno argumentovat, že původně plánovaná kružnicová událost nebyla zrušena, ale je v kalendáři uchována v modifikované podobě - sousední oblouk, který byl dřívější místní událostí rozseknut, se nahradí jeho částí, která je přilehlá ke zvýrazněnému střednímu oblouku. Povšimněte si, že původní soused i jeho část mají stejné ohnisko a proto původní a pozměněná kružnicová událost mají stejnou kružnici a tudíž jsou naplánovány na stejnou dobu (t.j. pro stejnou polohu zametací přímky).

V příkladech 4 a 5 je jeden ze sousedů zvýrazněného středního oblouku eliminován kružnicovou událostí, ke které došlo dříve. V tomto případě se původně plánovaná kružnicová událost nahradí jinou, ve které je stejný střední oblouk určený k eliminaci, ale je jiný jeden z jeho sousedních oblouků. Náhrada je podstatná, nový soused má jiné ohnisko a proto nová kružnicová událost nastane jindy, při jiné poloze zametací přímky.

V příkladech 6 a 7 dojde k místní události speciálního typu, kdy degenerovaná parabola dosaženého místa protne bod, ve kterém se zvýrazněný střední oblouk stýká se sousedním obloukem na pobřežní čáře. I v tomto případě dostane zvýrazněný střední oblouk zcela jiného souseda s jiným ohniskem a proto i kružnice nové události bude zcela jiná a k události dojde jindy, než bylo původně plánováno.

Dojde-li tedy k místní nebo kružnicové události, je nutné nejen naplánovat případné nové kružnicové události, ale také zkontrolovat zda se tím nenaruší některá z již naplánovaných kružnicových událostí.

Scéna: *Kalendář událostí*

Pro plánování výpočtu si algoritmus vede *kalendář událostí*. V předchozích scénách jsme implicitně předpokládali, že výpočet probíhá on-line a informace o místech se stává dostupná, když na ně narazí zametací přímka. Pokud je ale problém řešen off-line, je již na začátku výpočtu známa úplná množina míst. V takovém případě mohou být místní události vloženy do kalendáře událostí již na začátku, řazeny podle y-ových souřadnice odpovídajících míst, které udávají, kdy na ně zametací přímka narazí. Jsou do kalendáře vloženy hned na začátku výpočtu.

Kružnicové události se zařazují vždy, kdy dojde na pobřežní čáře ke změně, v důsledku které je možno očekávat, že ke kružnicové události dojde. Obsahují údaje o třech zúčastněných obloucích, jejich ohniscích, poloměru a středu kružnice a jsou do kalendářové fronty řazeny podle y-ové souřadnice nejvyššího bodu kružnice, který udává polohu zametací přímky v okamžiku, kdy k nim dojde.

Kalendář je na obrazovce znázorněn segmenty, nakreslenými při pravé straně okna. Černé segmenty odpovídají místním událostem; hrot na levé straně segmentu je ve stejné výšce jako je odpovídající místo (a udává tedy polohu zametací přímky, při které k události dojde). Klepne-li se na kalendářový segment, graficky se zvýrazní a zvýrazní se také jemu odpovídající místo. Totéž nastane, pokud klepneme na příslušné místo.

Fialové segmenty odpovídají kružnicovým událostem; hrot ukazuje na nejvyšší bod odpovídající kružnice a udává proto také polohu zametací přímky, při které k události dojde. Při klepnutí na segment se segment graficky zvýrazní a objeví se také odpovídající kružnice a zvýrazní se oblouk, který by měl při události zmizet. Totéž se stane při klepnutí na tento oblouk.

Jak již bylo uvedeno, i při místní, i při kružnicové události se může stát, že je některé naplánované kružnicové události třeba zrušit, protože zmizel některý z oblouků, který se události zúčastňuje buď jako utlačovatel nebo jako utlačovaný. Naopak se nový oblouk může objevit (v okamžiku události jako degenerovaný) nebo při vymizení oblouku se objeví nová trojice po sobě jdoucích oblouků, což může vést k nutnosti naplánovat novou kružnicovou událost. V okamžiku, kdy k události dojde, budou znázorněny jak rušené, tak i nově plánované události - pro odlišení budou první z nich zbarveny šedě, kdežto druhé zeleně. Těsně po události pochopitelně šedé kalendářové segmenty zmizí a zelené dostanou svou normální fialovou barvu kružnicové události.

Tato scéna tedy ilustruje, jak se mění kalendář událostí a jak podle něho algoritmus pracuje.

Často se stane, že události nastávají tak těsně po sobě, že výše uvedeným způsobem znázorněný kalendář je nepřehledný, protože se jednotlivé segmenty překrývají. Pak je možné vypnout volbu **Přesná poloha** a kalendářové segmenty se vertikálně přemístí tak, aby se nepřekrývaly, čímž ale dojde k tomu, že jejich poloha nebude odpovídat poloze zametací přímky v okamžiku, kdy k nim dojde (a proto také u nich nebudou ukazovány hroty na levé straně). Korespondenci místních událostí s místy a kružnicových událostí s kružnicemi a oblouky je ale možno určit volbou kalendářového segmentu nebo místa či oblouku myší, tak jak bylo uvedeno výše.

Scéna: *Degenerovaná událost*

Tato scéna umožňuje ukázat 8 různých konfigurací, při jejichž zpracování dochází k zvláštním situacím, které nazýváme degenerované. Konfigurace se volí na ovladači volbou **Příklad xx.** Po provedení volby používejte knoflík **Vpřed** pro animaci výpočtu, který se zastavuje na všech událostech.

V Příkladu 0 existují dvě nejjižnější místa. Spusťte animaci. Když zametací přímka dorazí do těchto míst, bude pobřežní čára tvořena dvěma navzájem oddělenými rovnoběžnými polopřímkami směřujícími k jihu, ale neprotínajícími se. Po posunu zametací přímky o infinitezimálně malý kousek k severu se z polopřímek stanou dvě standardní paraboly, jejichž *jediný* průsečík se rychle přibližuje z nekonečna, jak zametací přímka postupuje k severu. Toto je jediný případ, kdy je polopřímka patřící do Voroného diagramu kreslena "od nekonečna". Při programování je třeba tento případ ošetřit zvlášť. V příkladu je kreslení polopřímky ukončeno kružnicovou událostí krátce poté, kdy zametací přímka narazí na další místo.

Když v Příkladu 1 zametací přímka narazí na nejsevernější místo, pak dojde současně ke dvěma událostem: místní událost nalezení zmíněného místa a kružnicová událost, při které se setkají dva oblouky pobřežní čáry. Tyto dvě události se navzájem neovlivňují a mohou být zpracovány postupně v

libovolném pořadí.

V Příkladu 2 proběhne místní událost, která odpovídá nejsevernějšímu místu, zvláštním způsobem. Polopřímka spuštěná z tohoto místa (degenerovaná parabola) neprotne žádný ze dvou oblouků pobřežní čáry, ale strefí se přímo do jejich průsečíku. V takovém případě končí v průsečíku segment, který byl kreslen průsečíkem zmíněných oblouků a začnou se kreslit dva nové segmenty Voroného diagramu, určené průsečíky parabolického oblouku nového místa s původními dvěma oblouky. Tato událost musí být ošetřena zvláštním způsobem.

Příklad 3 je obdobný, ale současně s místní událostí pro severní místo dojde ke kružnicové události. V tomto případě by měla být nejprve zpracována kružnicová událost a oblouk odpovídající nejjižnějšímu místu odstraněn z pobřežní čáry a pak teprve událost místní.

V Příkladu 4 dojde současně ke dvěma kružnicovým událostem a zmizí dva sousední oblouky. Události lze zpracovat postupně v libovolném pořadí.

V Příkladu 5 leží všechna místa na jedné přímce. Takováto konfigurace, která musí být ošetřena zvlášť, vede k tomu, že Voroného diagram bude tvořen $n-1$ rovnoběžnými přímkami (kde n je počet míst), které rozdělí rovinu do dvou polorovin oddělených $n-2$ pásy nekonečné délky.

Příklad 6 je podobný, ale všechna místa mají stejnou y-ovou souřadnici. Voroného diagram bude opět tvořen $n-1$ rovnoběžnými přímkami, dvěma polorovinami a $n-2$ pásy. Tato konfigurace, která je zobecněním Příkladu 0, musí být ošetřena zvlášť a jinak než v předchozím případě.

Nakonec Příklad 7 sice nepřináší nic nového, ale dojde v něm současně k 9 kružnicovým událostem, při nichž zmizí 9 sousedících oblouků a současně k místní události, jejíž spuštěná polopřímka se strefí do bodu, kde zmizely oblouky. I přes svoji složitost se kombinovaná událost zpracuje snadno provedením kružnicových událostí v libovolném pořadí, následovaných místní událostí.

Scéna: *Vyhledávání oblouku*

Při místní události je třeba určit, který oblouk leží pod nově objeveným místem. V zásadě je tato úloha jednoduchá, i když se při jejím řešení musí vypočítávat dosti komplikované výrazy. Pobřežní čára se reprezentuje posloupností míst, která jsou ohnisky parabolických oblouků, ze kterých se skládá. Je dobré si uvědomit, že některé místo se může v posloupnosti vyskytovat vícekrát (několik oblouků patří téže parabole, ale mezi ně jsou vklíněny oblouky s jinými ohnisky).

V této scéně jsou míst aoznačena písmeny a posloupnost reprezentující pobřežní čáru je zapsána ve spodní části obrazovky. Klepnutí na oblouk zvýrazní odpovídající prvek posloupnosti a naopak. Povšimněte si, že velmi často

je řada oblouků pobřežní čáry mimo obrazovku (chcete-li je vidět, použijte zoom).

Jak již bylo řečeno, skoro vždy se alespoň některá ohniska objeví v posloupnosti vícenásobně. Zkuste si přiřazovat prvky posloupnosti obloukům dříve, než se o správnosti přiřazení přesvědčíte klepnutím na oblouk.

Jestliže známe polohy ohnisek oblouků pobřežní čáry a navíc polohu zametací přímky, je snadné pomocí metod analytické geometrie určit místo, kde se tyto oblouky dotýkají: ze znalosti ohniska a řídící přímky dostaneme snadno rovnici odpovídající paraboly. Průsečík leží na dvou parabolách současně a tedy vyhovuje dvěma rovnicím parabol najednou. Z toho již dostaneme kvadratickou rovnici a jejím řešením dostaneme souřadnice jejich průsečíků. Trochu nápaditosti je třeba vynaložit na to, abychom zjistili, které ze dvou řešení kvadratické rovnice je průsečík, který hledáme (viz následující scéna).

Máme-li pak posloupnost x-ových souřadnic průsečíků oblouků pobřežní čáry, seřazených zleva doprava, je jednoduché zjistit, mezi které dvě padne x-ová souřadnice nově nalezeného místa.

Jestliže je ale množina míst velká a je rozložena do šířky, může se stát, že pobřežní čára obsahuje velké množství oblouků. Pak by uvedený postup mohl být příliš pomalý. Jde-li nám o co nejrychlejší výpočet i za cenu větší logické složitosti algoritmu, je výhodné si nad pobřežní čárou vytvořit binární vyhledávací strom tak, jak je to ilustrováno v této scéně.

Listy stromu (oranžová kolečka) odpovídají obloukům čáry (každý list leží nad odpovídajícím obloukem) a vnitřní vrcholy stromu (šedá kolečka) odpovídají styčným bodům oblouků (v nákresu vnitřní uzel stromu leží vždy nad odpovídajícím průsečíkem oblouků). V každém vnitřním bodě je uložena informace o ohniscích oblouků, které se v odpovídajících styčných bodech potkávají (přičemž je poznamenáno, které ohnisko odpovídá levému oblouku a které pravému).

Hrany stromu jsou kresleny zeleně. Příslušnost vrcholů stromu obloukům nebo jejich průsečíkům je znázorněna oranžovými nebo šedivými svislými čarami (které nejsou součástí stromu, jen ilustrují logickou souvislost prvků pobřežní čáry s uzly stromu). Svislé čáry je možno skrýt volbou na ovladači, která umožňuje i skrýt celý strom.

Je nutné poznamenat, že tvar stromu je pro správnou funkci algoritmu nepodstatný. Množinou oblouků pobřežní čáry je pevně dána množina listů stromu, ale je lhostejně, jak se vybuduje struktura vnitřních vrcholů stromu a jejich napojení. Pro rychlost výpočtu ale bude vhodné, aby strom byl co nejlépe vyvážený.

Nyní si představme, že je dáno místo, na které narazila zametací přímka při místní události a jsou známy všechny parametry pobřežní čáry i stromu nad ní. Vyhledávání oblouku pod právě dosaženým místem začíná v kořenu stromu a v každém vnitřním uzlu pokračuje následujícím způsobem:

pro danou polohu zametací přímky (řídící přímky parabol nesoucích oblouky pobřežní čáry) se ze znalosti poloh ohnisek oblouků, nad jejichž průsečíkem se uzel nachází, určí rovnice parabol, kterých jsou oblouky částí. Rovnost rovnic parabol vede na kvadratickou rovnici, z níž se určí souřadnice styčného bodu oblouků. Porovnáním x-ových souřadnic styčného bodu a nového známého místa odkrytého při místní události se určí, zda ve stromě postupovat do levého či pravého syna (a ve vzácných případech se může stát, že nové místo je nad průsečíkem oblouků).

Uvedeným postupem v nedegenerovaném případě sestoupíme až do listu, který odpovídá hledanému oblouku.

Při místní události je do stromu třeba přidat dva listy a dva vnitřní vrcholy (při degenerované místní události po jednom) a při kružnicové události je zase třeba jeden list a jeden vnitřní vrchol odebrat. Tyto operace zde nebudeme podrobně rozebírat; zkuste si obvyklým způsobem pohybovat zametací přímkou a sledovat změny ve stromu nad pobřežní čárou.

Počet kroků, které je pak nutné provést při hledání oblouku pod nově dosaženým místem je tedy úměrný nikoli počtu oblouků pobřežní čáry, ale hloubce stromu, která v optimálním případě bude okolo logaritmu počtu oblouků pobřežní čáry. Tím se někdy dosáhne výrazného zrychlení výpočtu.

Může se ale stát, že se strom nad pobřežní čárou stane nevyváženým a jeho hloubka bude příliš veliká. Abychom tomu předešli, můžeme použít obvyklé způsoby vyvažování, které byly uvedeny v kapitole o stromových datových strukturách. V appletu však vyvažování pro jednoduchost použito není.

Scéna: *Průsečík oblouků pobřežní čáry*

Tato scéna je malou technickou odbočkou. Ukážeme si, jak zjistit průsečík dvou oblouků pobřežní čáry. Scéna ukazuje zametací přímku, danou jako rozhraní světlé a tmavé oblasti okna a dvě místa, označená červenou a zelenou barvou. Oblouky mají v této scéně stejnou barvu jako jejich ohniska. Předpokládáme, že jsou numericky dány polohy zametací přímky i obou míst.

Jak již bylo řečeno, pro každé z obou ohnisek si určíme snadno rovnici paraboly s daným ohniskem, pro kterou zametací přímka je řídící přímkou. Z rovnosti hodnot těchto parabol dostaneme kvadratickou rovnici, ze které vypočteme polohu průsečíku oblouků. Kvadratická rovnice má ale obecně dvě řešení a je třeba určit, které z nich je to pravé.

Na začátku je červené i zelené místo ve stejné výšce. To vede k tomu, že se odpovídající paraboly rozevírají shodně a výše uvedená kvadratická rovnice má jediné (dvojnásobné) řešení. Zde tedy není problém určit průsečík parabol.

Posuňte nyní myší červené místo k severu. Jím určená parabola nyní bude užší a z paraboly nižšího zeleného místa vysekne kus a do něho se vklíní. Zleva tedy bude napřed zelený oblouk, pak červený oblouk a nakonec znovu zelený oblouk. Pokud víme, že z průsečíku oblouků vychází oblouk zeleného ohniska

doleva a oblouk červeného ohniska doprava, pak průsečíkem je to řešení, které má menší x-ovou souřadnici. Pokud z průsečíku oblouků vychází oblouk červeného ohniska doleva a oblouk zeleného ohniska doprava, pak průsečíkem je to řešení, které má větší x-ovou souřadnici.

Posunujte nyní ohnisky vodorovně ale tak, aby červené bylo stále severněji. Uvidíte, že to, co bylo řečeno v předchozím odstavci je určeno tím, že červené ohnisko je více na sever, ale je lhostejno, zda je více na západ nebo na východ od zeleného ohniska.

Přemístěte nyní místa tak, aby bylo severněji zelené místo. Pak se role ohnisek prohodí - oblouk se severnějším ohniskem vždy vychází z průsečíku s menší x-ovou souřadnicí doprava a z průsečíku s větší x-ovou souřadnicí doleva.

Scéna: *Určování nejbližšího bodu*

Základní aplikací Voroného diagramu je problém najít pro danou množinu míst a zadaný bod x roviny to místo, které je bodu x nejbližší. Velice často je množina míst pevná a nemění se a dotazů na nejbližší místo se provádí velké množství. V takovém případě je vhodné informaci o množině míst předem zpracovat tak, aby potom dotazy na nejbližší místo byly zpracovávány co nejrychleji.

V takovém případě lze postupovat následujícím způsobem:

Nejprve se sestrojí Voroného diagram tak jak je ukázán na obrazovce a všemi koncovými body segmentů Voroného diagramu se proloží svislé přímky. Vyberte volbu **Svislé přímky** nebo stiskněte knoflík **Dále,** aby se tyto přímky znázornily.

Přímky rozdělí rovinu do svislých pruhů a pruhy jsou segmenty Voroného diagramu rozsekány do lichoběžníkových oblastí (v některých případech tento lichoběžník degeneruje na trojúhelník, v dalším ale budeme mluvit jen o lichoběžnících).

Vyberte volbu **Jeden pruh** nebo stiskněte knoflík **Dále** pro zvýraznění lichoběžníkových oblastí jednoho pruhu; klepnutím myší postupně volte různé pruhy a sledujte jejich rozdělení do oblastí. Nakonec zvolte **Všechny pruhy** nebo stiskněte knoflík **Dále** pro zvýraznění všech lichoběžníkových oblastí.

Způsob, jak byly lichoběžníkové oblasti zkonstruovány vede k tomu, že každá lichoběžníková oblast je *celá* částí spádové oblasti jistého místa a tedy všechny vnitřní body lichoběžníka mají totéž nejbližší místo. Totéž místo je i nejbližším místem bodů na obvodu lichoběžníka, na rozdíl od vnitřních bodů však nemusí (i když může) být jediným nejbližším místem.

Úsečky, které tvoří obvod lichoběžníka, mohou být uvnitř spádové oblasti jednoho místa a nebo být celé součástí segmentu Voroného diagramu, ve kterém se stýkají spádové oblasti dvou míst. V obou případech mají všechny body

vnitřku obvodové úsečky stejnou informaci o nejbližším místu nebo místech; takové místo je jedno nebo dvě.

Nakonec vrcholy lichoběžníků jsou buď Voroného body a nebo leží uvnitř Voroného segmentů. Pro každý takový bod tedy lze snadno určit dvě nebo tři nejbližší místa (výjimečně i více).

Pokud nás zajímá jen alespoň jedno nejbližší místo a nikoli přesná informace o všech nejbližších místech dotazovaného bodu, stačí si pro celý lichoběžník pamatovat jen základní nejbližší místo určené pro vnitřek.

Vytvořením lichoběžníků a informací o nejbližším místě nebo místech odpovídajících vnitřku lichoběžníka a částem jeho obvodu končí předzpracování.

Hledáme-li pak pro daný bod nejbližší místo, stačí určit lichoběžník, ve kterém leží a přečíst informaci s lichoběžníkem spojenou. K tomu si napřed určíme svislý pás, ve kterém leží, což lze provést velmi efektivně půlením. Předpokládejme, že $x_1, \ldots, x_\ell$ jsou x-ové souřadnice svislých přímek ohraničujících pásy a uspořádané vzestupně. Nejprve porovnáme x-ovou souřadnici dotazovaného bodu s číslem $x_{\ell/2}$; podle toho zda leží nalevo či napravo od mediánové svislé přímky souřadnici porovnáme s číslem $x_{\ell/4}$ nebo $x_{3\ell/4}$ atd. a po nejvýše $\lceil \log_2 \ell \rceil$ krocích identifikujeme pás, do kterého bod patří. Vzácně také můžeme zjistit, že bod leží na hranici dvou pásů.

Pro každý pás zvlášť je potom třeba vytvořit podobné dotazovací schéma, ve kterém namísto svislých přímek vystupují kosé segmenty, dělící pás do lichoběžníků. Dotazy jsou zde o něco komplexnější než v případě svislých přímek, ale půlením pásu (dle počtu lichoběžníků) lze hledaný lichoběžník nalézt opět v čase úměrném logaritmu počtu lichoběžníků v pásu.

Scéna: *Delaunayova triangulace*

Nakonec ještě ukážeme Delaunayovu triangulaci množiny bodů v rovině. Jedná se vlastně o duální graf k grafu představovaném Voroného diagramem. Stručně řečeno dvě místa se spojí modrou čarou, pokud jejich Voroného oblasti mají společnou stěnu (jeden společný bod nestačí).

Pokud Voroného diagram nemá body, ze kterých vychází čtyři nebo více segmentů diagramu, pak modré čáry Delaunayova diagramu rozdělí rovinu do trojúhelníkových oblastí a proto obvykle mluvíme o triangulaci.

Zatrháváním checkboxů **Voronoj** a **Delaunay** můžeme zobrazovat a skrývat oba diagramy. Zobrazte si je současně a ujistěte se, že byste uměli nakreslit Delaunayovu triangulaci na základě Voroného diagramu.

Tato scéna je uvedena spíše pro informaci, blíže konstrukci Delaunayovy triangulace, její vlastnosti a aplikace rozebírat nebudeme.

Scéna: *Původní Fortunův algoritmus*

V původním článku Steve Fortune popsal algoritmus hledání diagramu v jiné podobě. K jejímu objasnění si znovu ukážeme 3D situaci. Pohoří tvořené

kužely míst se nejprve ukáže tak jak tomu bylo ve výchozí poloze scény "Kužely", tedy při pohledu shora. Ihned se ale začne naklápět "hlavou dolů", ale pohyb se zastaví po naklopení o 45 stupňů. Nakonec se tedy na kužely díváme ve směru jižních spádnic kuželových hor (ale z nekonečna nebo alespoň velké dálky, abychom potlačili perspektivu).

Způsob pohledu na pohoří znázorňuje v rohu obrazovky nakreslená hlava. Zvolený úhel pohledu má jednu výhodu a jednu nevýhodu.

Výhodou je to, že se díváme ve směru zametací roviny, jak byla ukázána ve scéně "Zametací rovina", takže ta se jeví jako přímka, do které se promítá vše, co v rovině leží. Mezi jiným se do ní promítne i celá pobřežní čára, protože ta je vlastně průsečíkem zametací roviny a povrchu pohoří a jako taková tedy je součástí zametací roviny.

V původní verzi Fortunova algoritmu tedy nebyla oddělená zametací přímka a pobřežní čára a také odpadla mrtvá oblast mezi zametací přímkou a pobřežní čarou, která sice už ležela v "zametené" oblasti, ale ještě mohla být ovlivněna místy, kterých se zametací přímka ještě nedotkla. Jednalo se tedy o čistý zametací algoritmus, jak je znám z jiných problémů ve výpočetní geometrii.

Na druhé straně tvar segmentů a oblastí Voroného diagramu přestal být přímkový. Ve 3D zobrazení vidíme, že Voroného segmenty, oddělující oblasti jednotlivých míst ve skutečnosti nejsou úsečky, ale hyperbolické oblouky, vzniklé protnutím dvou kuželových ploch s rovnoběžnými osami. Jako úsečky se pouze jeví při pohledu shora, jak jsme to činili většinu této kapitoly. Původní Fortunův směr pohledu ale není svislý a hyperbolické křivky se nepromítají jako úsečky, ale jako skutečné křivky. Voroného oblasti míst jsou proto zvláštním způsobem zkreslené.

Důležité je, že jižní spádnice kuželů se jeví jako body a proto se každé místo jeví jako nejnižší bod své oblasti. Proto je libovolné místo prvním bodem celé své oblasti, na který zametací přímka narazí a tedy v okamžiku, kdy máme začít kreslit oblast nějakého místa, máme informaci o tomto místu k dispozici. Na rozdíl od toho při pohledu shora zametací přímka přechází přes velkou část Voroného oblasti dříve, než narazí na místo, které tuto oblast určuje.

Mohlo by se zdát, že popis výpočtu Fortunova algoritmu v původním provedení musí být mimořádně náročný, protože segmenty jsou tvořeny složitými křivkami. Ve skutečnosti ale potřebujeme znát pouze koncové body segmentů, které určíme při místních a kružnicových událostech a průběh křivky mezi nimi je nepodstatný. Na konci výpočtu se všechny koncové body segmentů přetransformují do svislého směru pohledu a propojí úsečkami.

Ovladač umožňuje snadno překlápět pohoří z vertikálního pohledu do Fortunova směru pohledu knoflíky **Vertikální** a **Fortune** pro objasnění vztahu původní a zde vykládané podoby algoritmu.

Část VI

Vyhledávání řetězců

Při zpracování textů je základní úlohou hledání daného řetězce znaků (vzoru) v daném textu. V kapitole budou popsány tři algoritmy pro vyhledávání.

První z nich, který navrhli Rabin a Karp, je uveden spíše pro zajímavost a proto, že je velmi jednoduchý a snadný na pochopení.

Na druhé straně algoritmus, který popsali Knuth a Pratt a nezávisle na nich Morris, patří k základním algoritmům pro vyhledávání. Algoritmus má dvě fáze; při předzpracování pracujeme pouze se vzorem a odvodíme z něho detailní informace, které potom ve fázi vlastního vyhledávání velmi zrychlují výpočet.

Poslední algoritmus je zobecnění algoritmu Knuth-Morris-Pratt na případ, kdy máme vzorů více. Bylo by sice možné každý vzor hledat zvlášť tak, že pro každý vzor zopakujeme výpočet znovu, ale Aho a Corrasick navrhli postup, při kterém vyhledáváme více vzorů najednou jen o málo pomaleji než při hledání jediného vzoru.

Kapitola 23

Algoritmus Rabin-Karpův

Scéna: *Hledání vzoru v textu*

V této kapitole se budeme zabývat hledáním všech výskytů jistého krátkého textu - vzoru - v dlouhém textu. Z důvodů, které vysvitnou později, budeme předpokládat, že abeceda z níž jsou vzor i text složeny jsou číslice 0,1,2,3,4,5,6,7,8,9. Tento předpoklad není příliš omezující; jakékoli symboly si můžeme zakódovat jako čísla a není podstatné, zda pracujeme s dekadickou soustavou jako zde, nebo se soustavou se základem 256 jako když jsou základem jednobytové znaky nebo s jiným základem.

Tato scéna jen ukazuje základní situaci, kdy se kolem nehybného jasně zeleného vzoru pohybuje dlouhý text zelenomodré barvy a v okamžiku, kdy se všechny číslice ve vzoru shodují s odpovídajícími číslicemi textu, se objeví upozornění na nalezení výskytu vzoru v textu.

Text i vzor je možno měnit: po klepnutí na kterýkoli box vzoru nebo textu (box změní barvu) je možno v něm vyplněnou číslici přepsat. Knoflíky ovladače je také možno vytvořit celý nový vzor nebo nový text (jeho délku lze také nastavit). Text je možno vytvořit náhodně nebo způsobem nazvaným "Zaručující vstup", viz volba na ovladači, kdy se do textu náhodně vsune předepsaný počet kopií vzoru (uvedený v poli **Výskytů)** a pak se náhodně doplní nedoplněné znaky. Výhodou druhého postupu je, že v textu se objeví podstatně větší množství výskytů vzoru, než pokud by byl generován náhodně.

Scéna: *Vzor jako číslo*

Základní myšlenkou algoritmu je dívat se na vzor i úsek textu, který se nachází proti vzoru, jako na číslo. Porovnávaný úsek textu je červený a je vytažen mezi vzor a text, aby tento pohled byl zdůrazněn. Jde tedy o to, kdy se zelený vzor a červené číslo rovnají. Na první pohled se nezdá, že by tato

myšlenka přinášela něco nového a užitečného, ale uvidíme, že se dá upravit tak, že dá užitečný a jednoduchý algoritmus.

Scéna: *Aktualizace čísla z textu*

Jde nám nyní o to, jak číslo z textu přepočítávat, když se text a vzor proti sobě posunou, abychom ho nemuseli znovu vytahovat celé z textu. Krok posunu bude nyní rozdělen do několika etap, které obvyklým způsobem krokujeme knoflíkem **Fáze:**

Krok: *Posun*

Text se posune proti vzoru a červené číslo vytažené z textu se posune s ním. ◇

Krok: *Vynásobení*

Červené číslo se doplní nulou, aby nejnižší řád vzoru (číslice napravo) ve vzoru měla proti sobě pravou číslici (zatím nulu) textového čísla.

Tato operace vlastně znamená vynásobit textové číslo základem poziční soustavy, v našem případě číslem 10. ◇

Krok: *Doplnění nejnižšího řádu*

Bledou barvou nakreslená nula se nahradí číslicí z textu. Tato operace znamená číslici přičíst k červenému textovému číslu. ◇

Krok: *Odstranění nejvyššího řádu*

Odstraníme nejvyšší (levou) číslici v červeném textovém čísle, která nekoresponduje s žádnou číslicí vzoru. Tato operace znamená odečíst od textového čísla číslo $c \cdot 10^{\ell}$, kde c je číslice, která byla odebrána a ℓ je délka vzoru (počet jeho číslic). Pokud by byla používána jiná číselná soustava než dekadická, základ soustavy by nahradil číslo 10 ve vzorci. ◇

Nyní je textové číslo zcela aktualizováno a můžeme jej porovnat s číslem daným vzorem. Vzorec, podle kterého se tedy textové číslo mění je

$$T_{new} = B \cdot T_{old} + b - c10^{\ell},$$

kde B je základ použité poziční soustavy, T_{new} je nová hodnota textového čísla, T_{old} je její původní hodnota, c je číslice, která se ocitla mimo vzor, b je číslice která se nově objevila proti vzoru a ℓ je délka vzoru.

Při krátkém vzoru již dostáváme prakticky využitelný a relativně rychlý postup.

V případě dekadických čísel reprezentovaných v počítači jako 32-bitové binární číslo by ovšem měl mít vzor nejvíce 9 znaků, protože 2^{32} se rovná 4.294.967.296, což je často málo. Použití 64-bitových čísel by přípustnou délku vzoru zvýšilo na maximálně 19 a i to může být nedostatečné.

Scéna: *Modulární porovnávání*

Je-li vzor dlouhý, nebylo by možno zaznamenávat vzor a textové číslo jako jedinou proměnnou typu `integer`. Bylo by nutné budovat aritmetiku delších čísel softwarově, což by algoritmus zkomplikovalo i citelně zpomalilo. Budeme proto zaznamenávat nikoli čísla samotná, ale jejich zbytky při dělení jistým pevným a rozumně velkým číslem. Otázku volby tohoto čísla ještě rozebereme později.

V této scéně je uvedené číslo, kterému budeme říkat *modulo*, za začátku rovno 31, ale je možné jej změnit.

Kromě vzoru a textového čísla jsou v této scéně znázorňovány i jejich zbytky při dělení zvoleným modulem. Nyní nastávají tři možnosti:

- zbytky odpovídající vzoru a textu se nerovnají; v tomto případě se nemohly rovna ani výchozí čísla, to jest vzor se tedy v této poloze s odpovídající částí textu neshoduje;
- vzor a textové číslo se rovnají; rovnají se proto i jejich zbytky při dělení modulem;
- vzor a textové číslo se sice nerovnají, ale jejich zbytky při dělení modulem jsou stejné.

Pokud tedy se zbytky nerovnají, můžeme postoupit dále, výskyt vzoru nebyl zjištěn. Jestliže se ale zbytky shodují, nejsme schopni odlišit druhou možnost od možnosti třetí, kterou budeme nazývat falešný poplach. V takovém případě je pak třeba porovnat vzor s textem znak po znaku.

Algoritmus je proto obzvláště výhodný v případech, kdy je počet výskytů vzoru v textu malý. Jestliže se podaří minimalizovat i počet falešných poplachů, velká většina poloh vzoru proti textu se probere velmi rychle a nutnost dodatečné kontroly v případě druhé a třetí možnosti uvedené výše již dobu výpočtu příliš neovlivní.

Zkuste si především projít text za použití původního modula 31 nebo jiného podobné nebo mírně větší velikosti, pak za použití velmi malého modula, například 3 nebo 5 a nakonec za použití velmi velkého modula řádů tisíců nebo i milionů. Uvidíte, že malé modulo sice přináší malé zbytky, se kterými se snadno a rychle operuje, ale také velké množství falešných poplachů, zatímco vhodně volené modulo řádu několika milionů způsobí, že k falešnému poplachu prakticky nikdy nedojde. K falešnému poplachu totiž dojde, pokud je rozdíl mezi vzorem a textovým číslem dělitelný modulem, a to je u velkého modula málo časté.

Při praktickém používání je tedy výhodné používat jako modulo číslo, které je svou velikostí blízké horní mezi velikosti celých čísel, které je v používaném počítači možno používat.

Pokud označíme modulo jako M, pak výše uvedený vzorec pro přepočítávání textového čísla lze upravit následujícím způsobem

$$Z_{new} = (B \cdot Z_{old} + b - cL) \bmod M,$$

kde Z_{new} a Z_{old} jsou nový a starý zbytek, B, b a c jsou číslice jako bylo uvedeno výše a $L = 10^{\ell} \bmod M$ je zbytek při dělení 10^{ℓ} modulem M. Bez ohledu na to, jak dlouhý je vzor a tedy jak velké bude textové číslo, jsou všechna čísla v tomto vzorci menší než M a tedy rozumně velká.

Scéna: *Úplný algoritmus*

Tato scéna umožňuje zopakovat si vše, co bylo řečeno výše.

Kapitola 24

Vyhledávací algoritmy Knuth-Morris-Prattův a Aho-Corasickové

24.1 Knuth-Morris-Prattův algoritmus

Scéna: *Hledání vzoru v textu*

V této scéně je text zapsán v zeleném pruhu a nad textem je zobrazen vzor, což je opět posloupnost písmen abecedy. Pro účely výkladu je výhodné nechat kurzor i vzor pevný a textem pohybovat vlevo a vpravo pomocí knoflíků **Krok** a **Zpět.**

Znaky textu se čtou zleva doprava a kurzor tvořený dvěma červenými trojúhelníky označuje, kam až byl text přečten. Okénka textu vlevo od kurzoru jsou světle zelená a obsahují znaky, které již byly přečteny, okénka vpravo od kurzoru jsou tmavě zelená a obsahují dosud nepřečtené znaky. Nepřečtené znaky záměrně barevně splývají s pozadím, aby bylo naznačeno, že je algoritmus ještě nezná.

Jestliže se některý znak vzoru shoduje s pod ním ležícím znakem v textu, jeho pozadí zežloutne. Hledáme *všechny* polohy vzoru, pro které se všechny jeho znaky shodují s odpovídajícími písmeny textu. V takovém případě vzor zbělá a objeví se upozornění na shodu. [Sorry: *Upozornění na shodu není zatím naprogramováno*]

Projděte vzorem podél celého textu a určete všechny jeho výskyty.

Jako znaky v textu i vzoru se používají velká písmena anglické abecedy. Text je možno změnit knoflíkem **Nový text.** Nový text se vytvoří v závislosti

na volbě na ovladači jako zcela náhodná posloupnost symbolů abecedy (v takovém případě je ovšem při delším vzoru málo pravděpodobné, že se vzor v textu alespoň jednou vyskytne) a nebo jako vstup se zaručenými výskyty vzoru. Volba mezi těmito možnostmi se provádí příslušnou volbou na ovladači.

V případě zaručených výskytů se do textu nejprve do náhodných poloh tolikrát zapíše vzor, kolik je číslo v poli **Vzorů** a potom se nedoplněné polohy v textu náhodně doplní symboly abecedy. (Vpisuje-li se více kopií vzoru, mohou nové kopie přepsat staré, a proto skutečný počet výskytů může být menší než bylo požadováno. To nastane například vždy, kdy by se požadovaný počet výskytů vzoru do textu dané délky bez překrývání nevešel. Na druhé straně může zcela výjimečně nastat i případ, že náhodně na doplnění přidané znaky vytvoří další kopie vzoru).

Libovolný znak vzoru nebo textu se také může změnit tak, že se políčko, ve kterém se nachází, aktivuje myší (změní barvu na fialovou) a pak se klepne na libovolnou klávesu představující písmeno (malá a velká písmena se nerozlišují a zobrazují se jako velká, nepísmenové klávesy jsou neaktivní).

Knoflík **Nový vzor** vytvoří nový vzor s náhodně generovanými znaky. Pokud jsou požadovány výskyty vzoru v textu, současně s vytvořením nového vzoru nebo přepsáním některého jeho znaku se vytvoří i nový text.

Zkuste si různé varianty vytváření textu a vzoru a pro každé nastavení také hledání výskytů vzoru v textu.

Scéna: *Plovoucí vzor*

Pokud by se hledaly výskyty vzoru tak, že pro všechny jeho polohy vzhledem k textu se porovnají všechny jeho znaky s odpovídajícími znaky textu, byl by výsledkem pomalý a neefektivní algoritmus. Cílem tohoto appletu je ukázat rychlejší a úspornější algoritmus, který nezávisle objevili Knuth s Prattem a Morris. Tato scéna zatím pouze představuje grafické prostředky, které budou použity pro jeho výklad.

S textem se zde manipuluje stejně jako ve scéně předchozí, ale vzor je zobrazován jinak. Na počátku je vzor nad nejlevějšími nepřečtenými znaky. Jelikož pole vzoru jsou nad nepřečtenými znaky, algoritmus nemůže zjistit, zda se shodují se znaky textu, které jsou pod nimi a proto se barva podkladu těchto polí nemění (i kdyby se dodatečně zjistilo, že tam došlo ke shodě).

Přečtení nového znaku v textu je obecně rozloženo do dvou fází, které ale budou někdy provedeny najednou.
čtecí fáze přečte nový znak textu (nejlevější z nepřečtených) a zobrazí jej na červeném pozadí,
dokončovací fáze převede právě přečtený znak k přečteným (a změní v souvislosti s tím barvu jeho pozadí). Přečtení znaku je znázorněno tak, že se text posune o 1 pole doleva, takže se pole s přečteným znakem přesune nalevo od kurzoru.

Ve čtecí fázi bude červené pole textu, ve kterém je čtený znak, nazýváno *aktivní*.

Vzor není nepohyblivý jako v předchozí scéně, ale může se podél textu přesně vymezeným způsobem posouvat. Jak již bylo řečeno, jeho základní poloha je ta, že jeho nejlevější znak se kryje s nejlevějším nepřečteným znakem (takže se nalevo opírá o kurzor).

Ve čtecí fázi se poloha vzoru nemění, ale jestliže se v průběhu čtecí fáze zjistí, že právě přečtený znak se *shoduje* s nad ním ležícím znakem kurzoru, pak se v následující dokončující fázi vzor posune o jedno pole vlevo také (je "zachycen" textem). Ta část vzoru, která vyčnívá vlevo od kurzoru se tedy vždy *shoduje* s textem pod ním. V krajním případě, kdy se celý vzor posune do oblasti vlevo od kurzoru, to znamená, že byl nalezen výskyt vzoru v textu.

Dobře si povšimněte, že znaky v té části vzoru, která je nalevo nad již přečteným textem představují současně *prefix* vzoru (tedy jeho počáteční úsek), ale také *suffix* přečteného textu (tedy jeho koncový úsek).

Jestliže se v průběhu čtecí fáze zjistí, že právě přečtený znak se *neshoduje* s nad ním ležícím znakem vzoru, pak v dokončující fázi vzor "sklouzne" vpravo tak, aby se všechny znaky vzoru, které leží vlevo od kurzoru, shodovaly s pod nimi ležícími znaky textu. Pokud je možností takového sklouznutí více, zvolí se nejkratší sklouznutí, neboli nejdelší možné vysunutí vzoru vlevo od kurzoru, při kterém je vysunutá část vzoru shodná s textem pod ním.

Za sklouznutí se považuje i to, že vzor, částečně vysunutý doleva, zůstane v dokončovací fázi stát, zatímco se text posune vlevo, protože i to je změna relativní polohy textu a vzoru.

Cílem této scény není ukázat, jak se zjistí, jak dlouhé má sklouznutí vzoru být. Výpočetně (i když ne myšlenkově) jednoduchý způsob zjišťování délky nejkratšího sklouznutí je podstatou Knuth-Morris-Prattova algoritmu a bude podrobně popsán v následujících scénách. V této scéně určování délky sklouznutí provádí Algovize. Možná se vám podaří na princip algoritmu přijít již nyní; pokud ne tak nezoufejte, opravdu to není zcela prosté.

Velmi často se ovšem stane, že po posunutí textu v souvislosti s přečtením znaku pro žádnou polohu vzoru vyčnívajícího vlevo přes kurzor nenastává shoda všech vyčnívajících znaků vzoru s odpovídajícími přečtenými znaky textu. V tomto speciálním případě pak musí vzor sklouznout do základní polohy, ve které je vpravo od kurzoru zarovnán proti dosud nepřečteným znakům.

Stav výpočtu bude číslo, které v každém okamžiku výpočtu udává, o kolik políček je vzor posunut proti své základní poloze, tedy kolik jeho polí je nalevo od kurzoru nad již přečtenými znaky textu. Je to tedy celé číslo v rozmezí mezi 0 a délkou vzoru (včetně těchto krajních mezí). Vzor je v textu nalezen právě tehdy, když stav výpočtu je roven délce vzoru a nalezený výskyt je pak na obrazovce nakreslen v textu vlevo od kurzoru. V této scéně i v několika

dalších je stav výpočtu numericky udán na obrazovce.

Zkuste si nyní znovu vyhledávání vzoru a podrobně sledujte, jak se vzor "zachytává" textu, pokud dochází ke shodě se čtenými znaky textu a jak "sklouzává" vpravo do nejbližší shody s textem nebo, velmi často, do své základní polohy.

Scéna: *Prefixy vzoru*

V této scéně je vzor zobrazen nikoli jednou jako v předchozích scénách, ale v několika kopiích (o 1 více než je počet znaků vzoru) v různých horizontálních polohách. Spodní kopie se celá nachází nad dosud nepřečteným textem a dotýká se na své levé straně linie kurzoru. Každá další kopie je o jednu úroveň výše a o jedno pole vlevo, takže nejvyšší kopie je celá nad přečteným textem a dotýká se linie kurzoru svou pravou stranou. Na rozdíl od minulé scény jsou kopie vzoru nepohyblivé.

Povšimněte si, že kopie vzoru jsou proti textu v právě všech polohách, ve kterých se mohl nacházet plovoucí vzor v minulé scéně.

Pro usnadnění popisu zavedu následující názvosloví: uvažujeme-li některou kopii vzoru, pak soubor jejích polí, které se nacházejí nad již přečteným textem, bude označován jako *prefix* a pole, které následuje bezprostředně vpravo od prefixu (a tedy leží nad znakem textu, který bude čten v nejbližší čtecí fázi) budeme nazývat *následné* pole příslušného prefixu. Znak v následném poli budeme nazývat *následný* znak. (Obecně se prefixem nazývá jakákoliv souvislá část kopie vzoru, která začíná prvním znakem vzoru, ale zde budu výraz prefix používat pouze tak, jak bylo uvedeno).

Prefixy jsou modré nebo bílé, následná pole prefixů jsou šedá a zbývající části kopií vzorů jsou skoro černé. Délka vyobrazených prefixů roste od 0 (spodní prefix) až po délku vzoru (horní prefix, který je přímo roven vzoru). Postoupíme-li nahoru o jednu vrstvu, zvýší se délka jejího prefixu o 1.

Jak jste si jistě povšimli, prefix má bílé pozadí právě když se všechny jeho znaky shodují s textem pod ním. Takový prefix budeme nazývat *shodný*. Prefix, který není shodný, má celý modré pozadí (bez ohledu na to, že některé - ale ne všechny - jeho znaky se mohou shodovat s odpovídajícími znaky textu).

Nejníže položený prefix (s nulovou délkou) je shodný *vždy* a to z triviálních důvodů: neobsahuje žádné pole, které by se případně mohlo neshodovat s textem.

Nyní velmi důležité pozorování, které dává do souvislosti předchozí scénu s plovoucím vzorem a tuto scénu. Kopie vzoru, ve které leží nejdelší shodný prefix, je v dané situaci horizontálně umístěna stejně jako v minulé scéně plovoucí vzor; délka nejdelšího shodného prefixu je rovna stavu výpočtu. Tato kopie vzoru je vyznačena ukazatelem. Projděte si znovu hledání vzoru v textu a zaměřte se na sledování této korespondence. V následující scéně se budeme dále a podrobněji tímto pohledem na činnost algoritmu zabývat.

Scéna: *Rozšiřitelná shoda*

Z výpočetního hlediska je neúnosné v každém kroku znovu určovat, které prefixy jsou shodné, porovnáváním všech políček všech prefixů s odpovídajícími políčky textu. Jestliže však známe všechny shodné prefixy v jistém okamžiku vyhledávání, je snadné a rychlé zjistit, které prefixy budou shodné v následujícím kroku.

Jak již bylo řečeno výše, je z triviálních důvodů vždy shodný prefix délky 0. Všechny další shodné prefixy se určí na základě následujícího jednoduchého pozorování.

Na obrazovce jsou stejně jako v předchozí scéně nakresleny kopie vzorů umístěné stupňovitě nad sebou a jejich prefixy, ležící nad již přečteným textem jsou opět bílé nebo modré podle toho, zda jsou nebo nejsou shodné. Následné znaky prefixů jsou šedé. Ve čtecí fázi ale barevné označování následných polí prefixů pozměníme: pole, které obsahují znak shodný s čteným znakem textu (znak v aktivním červeném poli), zežloutnou, ostatní zůstávají šedé.

Jestliže nyní je ve čtecí fázi prefix délky k shodný (tedy nakreslen bíle) a ve čtecí fázi zjistíme, že jeho následný znak je shodný s právě čteným znakem (což je označeno žlutou barvou pole, ve kterém se nachází), pak je po dokončení bezprostředně následující dokončovací fáze shodný prefix délky $k + 1$, což je vlastně původní prefix délky k doplněný o žluté pole. Právě popsaný jev je při animaci podtržen tím, že bílý shodný prefix délky k s následujícím žlutým polem se posunou šikmo doleva vzhůru do polohy $(k + 1)$-ního prefixu.

Projděte si algoritmus znovu a sledujte změny shodnosti prefixů v průběhu výpočtu.

Scéna: *Předzpracování*

V předchozí scéně nás zajímalo příliš mnoho informace, kterou ve skutečnosti nepotřebujeme. Tato scéna ukazuje pouze *stav* výpočtu, který byl výše definován jako délka nejdelšího shodného prefixu, a ve čtecí fázi také čtený znak. Automaticky také předpokládáme, že je znám vzor. Naopak předpokládáme, že text znám není - na počátku jsou textová pole prázdná. Prefixy vzorů jsou tmavě šedé - tato barva označuje, že jsme se dosud nepokusili určit, které z nich jsou shodné a které nikoli. Krokování v této scéně knoflíkem **Další** neukazuje výpočet, ale dedukci, která ukazuje postupně informace, jež je z těchto údajů možno odvodit.

Známe-li stav, víme, který je nejdelší (neboli nejvýše nakreslený) shodný prefix. Po prvním stisknutí knoflíku proto zbělá prefix v té kopii vzoru, která je určena stavem výpočtu. Delší prefixy (pokud existují) se zbarví tmavě modře, protože o nich zase víme, že shodné nejsou, protože jsou delší než udává stav výpočtu.

Prefix udaný stavem výpočtu zbělal, protože víme, že je shodný s textem. To nám ovšem říká, jaké jsou v takovém případě v textu symboly, které leží

pod bílým prefixem. Po druhém stisknutí knoflíku se proto tyto symboly v textu objeví.

Znalost symbolů, které se objevily v textu, nám také umožní jednoznačně určit, které z prefixů kratších než je stav výpočtu jsou shodné a které ne, protože známe všechny symboly textu, které pod nimi leží. Po dalším klepnutí knoflíku se tedy dosud tmavě šedé kratší prefixy zbarví bíle nebo modře podle toho, zda jsou nebo nejsou shodné.

Jestliže tedy je znám vzor a stav výpočtu, lze určit, které kopie vzoru se částečně shodují s textem, aniž by o textu bylo cokoli známo. Toto je základní pozorování, na kterém je založen popisovaný algoritmus.

Předpokládejme nyní, že známe nejen stav výpočtu, ale i symbol, který je přečten v čtecí fázi. Klepněte na knoflík **Další** po čtvrté a v aktivním místě textu se objeví písmeno. V tento okamžik je možno určit pro všechny kopie vzoru, zda se obsah jejich následného pole shoduje s přečteným červeným znakem textu, který je přímo pod nimi. Klepněte na knoflík po páté a následná pole obsahující shodné písmeno zežloutnou.

V tento okamžik jsme již schopni určit, u kterých shodných prefixů se shoda dá rozšířit o přečtený symbol. Jsou to prefixy, které jsou bílé a vpravo od sebe mají žluté pole. K nejdelšímu z nich se po dalším klepnutí přesune šipka, která dosud označovala prefix udaný stavem výpočtu. Klepněte ještě jednou a animovaným způsobem se provede prováděcí fáze a situace, kterou jsme předvídali, se explicitně zobrazí. Šipka označuje nový stav výpočtu, který byl určen pouze ze znalosti předchozího stavu výpočtu a přečteného symbolu.

Pokud tedy máme plnou informaci o vzoru, pak ze znalosti stavu výpočtu v jistém kroku a právě přečteného znaku textu lze určit stav výpočtu v následujícím kroku. Tuto funkci si v této scéně budeme tabelovat. Tabulka, která je na obrazovce, má řádky odpovídající stavům výpočtu (celá čísla v rozmezí od 0 do délky vzoru včetně) a sloupce odpovídají znakům abecedy (velká písmena A-Z anglické abecedy). Políčko v aktuálním řádku a sloupci je zvýrazněno žlutou barvou a po ukončení výše popsané dedukce se v něm objeví číslo stavu, do kterého přejdeme z aktuálního stavu daného řádkem tabulky, ve kterém je žluté políčko, při čtení symbolu daného sloupcem obsahujícím žluté políčko.

Po ukončení dedukce klepněte na jiné políčko tabulky a můžete celý postup opakovat. Předzpracování pro prohledávání, o kterém bude řeč v následující scéně, spočívá v doplnění celé tabulky tak, jak jsme to právě ukázali. Možná že vás správně napadne, že by se tabulka dala vyplnit s menší námahou, pokud bychom vhodným způsobem využívali hodnoty v některých políčkách tabulky pro určení dalších jednodušeji než opakováním právě probraného postupu. Algoritmus z následující scény ale ještě není cílový Knuth-Morris-Prattův algoritmus; v něm celá tabulka potřeba nebude a proto se nebudeme snažit její určení optimalizovat.

Scéna: *Konečný automat*

Tato scéna je malou odbočkou ve výkladu Knuth-Morris-Prattova (KMP); představuje postup, který je velmi rychlý, ale za cenu velkého plýtvání pamětí, pokud je abeceda, použitá pro napsání textu, většího rozsahu. Algoritmus KMP pak za cenu jen velmi mírného zpomalení výpočtu toto plýtvání zcela odstraní.

Výrazem "konečný automat" je míněna tabulka, která byla konstruována v předchozí scéně (čtenář, obeznámený s pojmem konečného automatu jistě bude vědět proč, ostatní ať si prostě místo "konečný automat" dosadí slovo "tabulka").

Postup má dvě fáze. První z nich je předzpracování, které vytvoří tabulku tak jak bylo popsáno v předchozí scéně. Jak bylo řečeno, k vytvoření tabulky stačí znát vzor, informace o textu není nutná. (To má výhodu v tom, že pokud vyhledáváme týž vzor častěji ve více textech, stačí předzpracování provést jen jednou.) V této scéně se pro daný vzor objevuje tabulka již hotová.

Cílem této scény je ukázat vyhledávání za pomoci konečného automatu, využívajícího sestrojenou tabulku. Projděte si výpočet; pak zatrhněte volbu **Skryj nepotřebné** a projděte výpočet ještě jednou. Nyní budete sice mít plnou informaci o hodnotách v tabulce, ale písmena vzoru nejsou zobrazena (veškerá potřebná informace o vzoru je v tabulce a tedy původní vzor už nepotřebujeme - zkuste přijít na to, jak vzor z tabulky rekonstruovat, pokud bychom ho třeba zapomněli) a z textu je vždy ukázán pouze čtený symbol (ten nám říká, do kterého sloupce se podívat). Uvidíte, že průběh stavu výpočtu je opravdu možno odvodit pouze z této informace, tedy znalosti tabulky a čteného symbolu.

Pokud je použitá abeceda velká (například Unicode), pak fáze předzpracování může být velmi dlouhá, protože tabulka má tolik sloupců, kolik je symbolů abecedy, ale jakmile je tabulka hotova, je vyhledávání odpovídajícího vzoru mimořádně rychlé: pro daný stav a čtený symbol se podíváme do tabulky a ihned vidíme, jaký bude následující stav. Celý výpočet spočívá tedy v tolika pohledech do tabulky, kolik má písmen text, ve kterém vyhledáváme. Přitom si pouze stačí uvědomit, že výskyt vzoru v textu je nalezen právě když stav výpočtu je dán číslem rovným délce vzoru.

Scéna: *Knuth-Morris-Prattův algoritmus - základní myšlenka*

Nevýhodou konečného automatu z předchozí scény je neefektivní použití velké tabulky, což je jednak plýtvání pamětí a také znamená mnoho práce ve fázi předzpracování. Na první pohled je patrné, že velká část hodnot v tabulce je rovna 0, což znamená, že se žádný shodný prefix nepodařilo rozšířit o další shodný znak. Jistě cítíte, že tabulku by bylo možno zkomprimovat. Knuth-Morris-Prattův algoritmus nevýhodu konečného automatu odstraňuje za cenu mírného zpomalení výpočtu.

Jak jsme viděli, určení stavu výpočtu v následujícím kroku výpočtu se ze znalosti současného stavu výpočtu a čteného symbolu provede takto: postupně se prochází shodné prefixy od nejdelšího (s délkou rovnou stavu výpočtu) až po nejkratší (s nulovou délkou) a pro každý se ověří, zda čtený symbol textu umožňuje shodu prefixu rozšířit o jeden znak. Jestliže alespoň jeden takový prefix existuje, pak nejdelší z nich určuje stav v následujícím kroku - ten bude o 1 větší než je jeho délka, protože se délka úseku shody prodlouží o 1. Na druhé straně pokud žádný shodný prefix se nedá rozšířit o čtený znak, pak bude v následujícím kroku stav 0.

Algoritmus pracující s konečným automatem tuto analýzu provádí ve fázi předzpracování; musí se proto připravit na všechny možné čtené symboly.

Knuth-Morris-Prattův algoritmus tuto činnost neprovádí před samotným probíráním textu v předzpracování vzoru, ale (pro úsporu místa) až při procházení textem. Aby ale výpočet nebyl příliš zpomalen, připraví si algoritmus dopředu, pouze ze znalosti vzoru a před samotným prohledáváním textu, pro každé celé k v rozmezí od 0 do délky vzoru včetně znak $G(k)$ a číslo $F(k)$, které budou neustále potřebné:

- $G(k)$ (pro k menší než délka vzoru) je znak, který musí být ve čtecí fázi přečten, aby se shoda prefixu délky k mohla rozšířit na shodu prefixu délky $k+1$, a
- $F(k)$ (pro $k > 0$) je délka nejdelšího z prefixů kratších než k, který je shodný, pokud je shodný prefix délky k.

Uvedené údaje jsou uvedeny v tabulce vpravo od kopií vzoru. Je jasně vidět, že $G(k)$ není nic jiného než $k+1$-ní symbol vzoru. Z dříve uvedené scény o předzpracování je jasné, že i funkci $F(k)$ můžeme určit pouze ze znalosti vzoru. V následující scéně si ale ukážeme, jak funkci F vypočítávat tak, aby bylo i předzpracování co nejrychlejší.

Knuth-Morris-Prattův algoritmus symboly G a F pro provedení jedné čtecí fáze používá takto:

Prochází shodné prefixy od nejdelšího (s délkou rovnou stavu výpočtu) po nejkratší (délky 0) a jakmile narazí na prefix, jehož shoda se dá rozšířit o čtený znak (tedy jeho následný znak je roven čtenému znaku), jako stav výpočtu určí o 1 zvýšenou délku tohoto prefixu. Pokud se na takový prefix nenarazí, bude následující stav výpočtu roven 0. Jelikož nejbližší kratší shodný prefix daného shodného prefixu se dá určit pomocí funkce F, uvedený postup se v řeči funkcí G a F dá přeformulovat takto:

položíme k rovno stavu výpočtu;
potom dokud $G(k)$ je různé od čteného znaku a $k > 0$ provádíme $k \leftarrow F(k)$;
po výskoku z cyklu v případě, že $G(k)$ je čtený znak, položíme nový stav výpočtu roven $k + 1$;
jinak nový stav výpočtu je 0.

Projděte si nyní výpočet algoritmu s tím, že tabulky G a F byly připraveny systémem, a sledujte jeho průběh.

Scéna: *Chybová funkce - výpočet*

V této scéně si ukážeme, jak se funkce G a F z předchozí scény výhodně určí. Lze to provést v etapě předzpracování, kdy známe vzor, ale neznáme text, ve kterém budeme jeho výskyty určovat.

Sloupec udávající funkci G se vyplní jednoduše: zdola nahoru se do něho vepíše vzor (a nejvyšší pole je prázdné). Je to tím, že tento sloupec je roven šedému sloupci následných znaků, který je v systému kopií vzoru na obrázku nakreslen *nad* čteným znakem textu a z toho, jak jsou prefixy uspořádány je zřejmé, že za prefixem délky k se nachází nad čteným znakem textu $(k+1)$-ní znak vzoru.

To, že lze i funkci F určit jen ze znalosti vzoru bez informace o textu plyne z úvahy, kterou jsme již provedli ve scéně o předzpracování pro prohledávání konečným automatem: pokud máme informaci, že prefix délky k je shodný, pak víme, že prvních k znaků vzoru je shodných s odpovídajícími k naposledy přečtenými vzory textu. Pak ale je možné určit pro každý prefix délky menší než k, zda je shodný a mezi nimi nalézt nejdelší; jeho délka pak bude udávat hodnotu $F(k)$.

Pro minimalizaci výpočetního úsilí při určování funkce F je však potřeba tuto základní myšlenku rozvinout vhodným způsobem. Hodnoty $F(k)$ budeme určovat postupně od nejnižších hodnot argumentu k, tedy v pořadí $F(0), F(1), F(2), \ldots$, takže když určujeme $F(k)$, známe již všechny předcházející hodnoty $F(0), \ldots, F(k-1)$.

Výpočet $F(k)$ je ilustrován na obrázku, kde jsou nakresleny kopie vzoru a v nich obsažené prefixy a jejich následné znaky tak, jak byly kresleny v předchozích scénách, dále bude nakreslen kurzor a svislice, která jím prochází (pro stanovení referenční polohy), ale nejsou zobrazeny znaky textu.

Scéna neukazuje výpočet funkce F od začátku, ale "z prostředka", pro $k = 5$, aby bylo možno lépe vyložit myšlenku výpočtu. Na začátek výpočtu je možno se vrátit opakovaným stisknutím knoflíku **Zpět** nebo přímo nastavit nulu do pole $K =$**.**

Výpočet hodnoty $F(k)$ sledujeme pomocí knoflíků **Další** a **Krok.** Prvních několik stisknutí knoflíku není vlastní výpočet $F(k)$, pouze se postupně objasňuje situace, ve které výpočet probíhá. Zde bude k dispozici knoflík **Další.** Na začátku předpokládáme, že k-tý prefix (označený šipkou) je shodný s textem.

První stisknutí knoflíku ukáže znaky textu, které na základě této informace lze určit. Je to k znaků, ležících nalevo od kursoru. To nám umožní určit, které prefixy kratší než k jsou také shodné. Po dalším stisknutí knoflíku se tyto prefixy zvýrazní.

Nyní se podíváme, jak situace vypadala o jeden krok vyhledávání dříve - po stisknutí knoflíku **Další** se text posune o jedno políčko zpět, tedy doprava. Informace o textu nyní říká, kterých bylo posledních $k-1$ přečtených znaků a který znak bude přečten. Všechny před stisknutím knoflíku shodné prefixy přejdou nyní na shodné prefixy o jedno políčko kratší, které mají své následné znaky (před chvílí ještě své koncové znaky) rovny znaku, který bude přečten. Obecně se také objeví další (v tento okamžik) shodné prefixy, jejichž shoda ale není rozšiřitelná o červený znak v textu.

Teď vidíme, jak určit $F(k)$: probíráme prefixy délky $F(k-1), F(F(k-1)), F((F(k-1))), \ldots$ neboli prefixy, které jsou v tento okamžik shodné a jsou kratší než $k-1$ tak dlouho, až narazíme na první, jež lze rozšířit o červený znak v textu. Jestliže ℓ je jeho délka, pak $F(k) = \ell + 1$. Povšimněte si, že používáme jen již známých hodnot funkce F. Od tohoto okamžiku máme k dispozici knoflík **Krok,** kterým si tento postup krokujeme. Žlutá šipka stále označuje prefix délky k nebo jeho předchůdce délky $k-1$, červená šipka ukazuje probírané prefixy. Nakonec se vrátíme do výchozí polohy textu a červená šipka udává hodnotu $F(k-1)$.

Jistě vám neuniklo, že výpočet hodnoty funkce $F(k)$ při předzpracování i operace Knuth-Morris-Prattova algoritmu pro jeden čtený symbol během samotného vyhledávání jsou si velmi podobny - v obou případech procházíme prefixy délek udaných iteracemi funkce F tak dlouho, dokud nenarazíme na rozšiřitelný prefix.

Scéna: *Knuth-Morris-Prattův algoritmus - výpočetní složitost*

Čtecí fáze v Knuth-Morris-Prattově algoritmu může být v některých případech rozložena do mnoha kroků, protože procházíme shora dolů shodné prefixy ve snaze nalézt takový, jehož shoda se dá rozšířit. Sledujte ale šipku, která stále ukazuje na shodný prefix, uvažovaný algoritmem.

V každé čtecí fázi nejprve šipka může (ale nemusí) skákat směrem dolů ke kratším prefixům a pak se může (ale nemusí) v následující dokončovací fázi posunout. Doba potřebná pro provedení celého prohledávání textu je úměrná počtu pohybů šipky, protože každý pohyb představuje jen provedení několika málo příkazů programu.

Nahoru se šipka může posunout nejvýše jednou v každé z dokončovacích fází a to pouze o jednu hladinu výše. Jelikož dokončovacích fází je tolik, kolik je znaků textu, součet délek posunů nahoru nemůže být větší než je počet znaků prohledávaného textu.

Počet skoků dolů ve čtecích fázích ale nemůže být větší než součet délek

posunů nahoru v dokončovacích fázích, protože šipka začíná v nejnižší hladině, jeden skok dolů je alespoň o jednu hladinu níže, ale šipka se nemůže dostat *pod* základní hladinu. Obrazně řečeno, máme-li provést několik skoků dolů, musíme vyšplhat dostatečně vysoko v krocích, jejich čtecí fáze nezahrnuje žádný skok, ale dokončovací fáze leze vzhůru.

Z toho plyne, že i počet posunů šipky nahoru, i počet jejích skoků dolu není větší než je délka prohledávaného textu a tedy doba, potřebná k prohledání textu je úměrná jeho délce.

24.2 Algoritmus Aho-Corasickové

Algoritmus Aho-Corasickové je zobecnění Knuth-Morris-Prattova algoritmu pro případ, že je dáno několik vzorů a hledáme najednou všechny jejich výskyty v textu. Pro delší text je algoritmus výrazně rychlejší než opakování prohledávání pro každý vzor zvlášť. Algoritmus může být užitečný například když podle něho pracuje server, který má dlouhý text ve kterém se vyhledává, a množství uživatelů mu posílá s vysokou frekvencí požadavky na vyhledání zvolených slov (vzorů). Pak může být z časových důvodů vhodné nehledat každý vzor zvlášť, ale vzory, které dorazily během jistého časového intervalu vyhledávat najednou dávkovým způsobem.

Scéna: *Vzory*

Vzory, které vyhledáváme v našem příkladu, jsou po řádcích napsány v levé části obrazovky. Jsou voleny tak, aby se ukázaly základní problémy, které při pro jejich vzájemné vztahy mohou nastat.

Z výkladu o Knuth-Morris-Prattově algoritmu již víte, že nás zajímalo, zda několik posledních znaků již přečteného textu nepředstavuje prefix vzoru. Takových prefixů mohlo být více najednou, ale každý z nich jinak dlouhý a tedy nejdelší z nich byl určen jednoznačně. Zde však, pokud by přečtený text končil znaky XFI, je takovým řetězcem I jako prefix slova IFINA, který má délku 1, a pak řetězec FI (délky 2), který ale je současně prefixem vzoru FIFINA i vzoru FINANA. Tato nejednoznačnost by komplikovala konstrukci vyhledávacího algoritmu.

Algoritmus Aho-Corasick si proto vzory uspořádá do stromu, který budeme nazývat *strom prefixů* (z důvodů vysvětlených v následující scéně) který se nachází v pravé dolní části obrazovky. Strom je zkonstruován tak, že cesty z kořene do vrcholů, které jsou tmavé, odpovídají navzájem jednoznačně vzorům, které mají být vyhledávány. Je tím míněno to, že posloupnost písmen, které přečteme, jdeme-li z kořene do daného tmavého vrcholu, je rovna některému vzoru. (Kořen jako jediný není označen písmenem - symbol λ, který je v něm zapsán je v souladu s běžnou praxí použit pro “nulový” nebo “prázdný”

znak; prázdný znak neuvažujeme.) Klepněte myší (levým knoflíkem) na některý tmavý vrchol stromu - zvýrazní se cesta do tohoto vrcholu z kořene stromu, ale také vzor, který této cestě odpovídá - zkontrolujte, že posloupnosti písmen jsou stejné. Naopak klepnete-li na některý vzor, zvýrazní se zvolený vzor, ale také odpovídající cesta do tmavého vrcholu ve stromu.

Povšimněte si, že všechny listy stromu jsou za všech okolností tmavé, tedy odpovídající vzorům. Normálně žádné vnitřní vrcholy tmavé nejsou - tak je tomu například v úvodním Příkladu 1. Pokud ale na ovladači zvolíte Příklad 2, uvidíte, že jestliže jeden ze vzorů prefixem jiného vzoru (v Příkladu 2 je vzor FIFI prefixem vzoru FIFINA), pak jemu odpovídající tmavý vrchol je (a musí být) vnitřním vrcholem stromu, který leží na cestě, odpovídající delšímu vzoru.

Příklad 3 jde ještě dále: dva ze vzorů jsou shodné. Takovým stejným vzorům pak odpovídá jediný (tmavý) vrchol stromu; když jej poklepete zvýrazní se shodné vzory oba. Tato nejednoznačnost nepřináší žádné zvláštní komplikace; vyhledávání je řízeno stromem a jen si poznačíme, že pokud najdeme v textu vzor odpovídající takovému tmavému vrcholu stromu, našli jsme vlastně vzory dva (nebo někdy i více).

Příklad 4 je naopak jednoduchý, obsahuje jeden vzor, V tomto případě se algoritmus Aho-Corasick redukuje na předchozí algoritmus Knuth-Morris-Pratt. Je zřejmé, že tvar stromu je triviální, takže pro jednovzorové vyhledávání nebylo třeba strom prefixů vůbec zavádět.

Nyní si také můžete vzory měnit a sledovat, jak se mění tvar odpovídajícího stromu. Pokud se na některé pole v tabulce vzorů klepne pravým knoflíkem myši, pole se aktivuje (změní barvu) a znak v poli lze přepsat. Místo napsání znaku je ale možno stisknout klávesu Delete, čímž se aktivní pole a s ním i všechna pole vpravo od něj vymažou, a pokud aktivní bylo nejlevější pole v řádce, je vzor ze seznamu odstraněn (pozor, Undo není implementováno).

Pokud se klepne pravým knoflíkem těsně vpravo od vzoru, do místa kde by se nacházelo případné další pole vzoru, pak se takové pole vytvoří jako prázdné (bez doplněného znaku) a současně se aktivuje, takže je možné znak vzápětí vepsat podle úvahy uživatele, čímž se současně vytvoří další prázdné aktivní pole a tak se pokračuje až do stisku klávesy Delete a nebo akce myší.

Podobně klepne-li se pravým knoflíkem myši pod první (nejlevější) pole spodního vzoru, vytvoří a aktivuje se první pole nového vzoru, který je pak možno podle potřeby dopsat. Pro dopisování jsou aktivní pouze klávesy představující písmena anglické abecedy (a Algovize malá písmena konvertuje na velká) a klávesa Delete se speciální výše popsanou funkcí.

Je také možno stisknout knoflík **Nové vzory** a Algovize vytvoří náhodně tolik vzorů, kolik je hodnota v poli **Počet vzorů** a jejich délka je náhodně volena až do maximální délky určené polem **Max. délka.** Náhodný soubor vzorů ale obvykle nedává příliš zajímavý strom, takže tato volba je spíš určena

pro vytvoření šablony, ve které pak znaky ručně přepisujeme.

Scéna: *Prefixy vzorů*

Tato scéna je velmi podobná předchozí, ale zajímají nás nyní prefixy vzorů a ne celé vzory. Klepněte myší (levým knoflíkem) na libovolné pole v tabulce vzorů nalevo. Zvýrazní se prefix vzoru ve zvoleném řádku, který začíná jeho nejlevějším polem a končí zvoleným polem. Současně se zvýrazní cesta z kořene stromu nakresleného vpravo do některého jeho vrcholu; je vybrána tak, aby znaky, které čteme, jdeme-li po ní od kořene dávaly zvolený prefix. Strom je sestrojen tak, aby taková cesta vždy existovala. Je zřejmé, že může být *jen jedna*, protože žádný vrchol stromu nemá dva různé syny nesoucí stejný znak. Může se stát, že zvolený prefix je zároveň prefixem jiného vzoru (např. když v Příkladu 1 zvolíte prefix FI). Takový prefix se pak zvýrazní také; je ale jasné, že oběma (nebo všem) odpovídá jediná cesta ve stromu. To je také důvod, proč se strom vytváří - je to vlastně komprese tabulky vzorů taková, že každý prefix libovolného vzoru se objeví v jednoznačně určené poloze.

Klepněte nyní na některý vrchol stromu: zvýrazní se cesta z kořene do zvoleného vrcholu a také se v tabulce vzorů, nakreslené vlevo, zvýrazní prefix (nebo prefixy) odpovídající zvolené cestě ve stromu. Každému vrcholu stromu odpovídá prefix alespoň v jednom vzoru (pochopitelně v rámci jednoho vzoru je jednoznačně určen délkou). Zvolíte-li ovšem kořen, pak jemu odpovídá prázdný prefix, jehož zvýraznění v tabulce vzorů není poznat.

Zkoušejte si korespondenci prefixů a cest ve stromu pro Příklady 1-4 i pro soubory vzorů, které si sami vytvoříte.

Jelikož tedy vrchol stromu a prefixy vzorů si navzájem jednoznačně odpovídají, budu v dalším trochu nepřesně o vrcholech stromu mluvit přímo jako o prefixech vzorů.

Jestliže s je vrchol stromu a a je symbol abecedy, pak pokud vrchol s má následníka s' označeného symbolem a, pak s' budeme označovat jako $G(s, a)$. Pokud takový následník neexistuje, bude výraz $G(s, a)$ nedefinován. Jestliže se na s díváme jako na prefix, pak je-li $G(s, a)$ definováno, znamená to, že s je vlastní prefix některého vzoru, který je možno prodloužit o symbol a na řetězec, který je také prefixem tohoto vzoru. Pokud tato situace pro žádný vzor nenastává, $G(s, a)$ není definováno.

Scéna: *Vytvoření stromu*

I když asi již tušíte, jak se strom odpovídající množině vzorů dá vytvořit, tato scéna to explicitně ukáže, jak je to možno udělat. Konstrukci si krokujte knoflíkem **Krok** nebo **Skok,** které mají i své odpovídající knoflíky **Zpět.** Poznamenávám, že se konstrukce dá provádět i jinak, např. probírat nejprve prefixy délky 1, pak 2, atd.

Na začátku je dána tabulka vzorů a ze stromu je nakreslen jen kořen, odpovídající prázdnému prefixu (prefix délky 0). Nejprve projdeme v tabulce první vzor a v podstatě jej jen přepíšeme do vytvářeného strumu jako cestu pověšenou pod kořenem.

Předpokládejme nyní, že jsme již prošli jeden nebo více vzorů v seznamu a začínáme procházet nový. Počínajíce od kořene stromu se pokoušíme ve stromu jít po cestě označené stejnými písmeny jako nacházíme ve stromu. Pokud se to daří, strom neměníme, jen v něm postupujeme dál. Jakmile ale z některého vrcholu nemůžeme postupovat dál do vrcholu označeného aktuálním písmenem ve vzoru, vytvoříme ve stromu novou větev, do které přepíšeme zbývající symboly vzoru. Do stromu se již nevracíme, i kdyby to dále bylo možné. Myslím, že konstrukce stromu je z appletu dostatečně zřejmá. Zkuste si konstrukci pro Příklady 1-4 i pro soubory vzorů dle vlastní volby.

Scéna: *Funkce OUT*

V této scéně je opět možno měnit množinu vzorů a Algovize ukazuje odpovídající strom prefixů. Vrcholy stromu (s výjimkou kořene) ale nyní obsahují tři pole; v prostředním je zapsán znak, tak jak tomu bylo v předchozích scénách. V levém poli jsou pořadová čísla vrcholů, používaná výhradně pro jejich identifikaci jako jejich jména.

Význam pravého pole je následující: jak jsme řekli, vrchol stromu lze považovat za prefix s některého vzoru nebo vzorů. V pravém poli je vypsáno pořadové číslo všech vzorů, které jsou sufixy řetězce s. Informace v pravém poli vrcholu s je obvykle označována jako $OUT(s)$.

V případě tmavého vrcholu je s dokonce *rovno* některému vzoru (a tedy je také jeho sufixem) a proto tento vzor je v $OUT(s)$ uveden. Pokud tabulka vzorů obsahuje několik identických vzorů, pak jsou všechny uvedeny v poli OUT odpovídajícího vrcholu stromu. Na druhé straně pole OUT světlých vrcholů stromu je obvykle prázdné.

V některých případech ale některý prefix s (bez ohledu na to, zda je roven vzoru, tedy odpovídá tmavému vrcholu nebo je vlastním prefixem vzoru, tedy odpovídá světlému vrcholu) zahrnuje jako vlastní sufix některý jiný vzor s'. V takovém případě vzor s' je zahrnut v $OUT(s)$.

Obecně proto $OUT(s)$ může zahrnovat referenci na více vzorů a pole $OUT(s)$ světlého vrcholu stromu může být neprázdné, i když tyto případy jsou spíše výjimkou.

Zkuste si opět sledovat tvar stromu a především funkci OUT pro různé soubory vzorů. Zkuste si především předdefinované případy; v některých OUT pro některé vrcholy zahrnuje více referencí. Zkuste si také takový soubor vzorů sami vytvořit.

Strom prefixů doplněný o funkci OUT obsahuje všechnu potřebnou informaci o množině vzorů. V dalších scénách sice vzory budeme stále zobrazovat,

ale jen pro informaci a pracovat budeme pouze se stromem.

Scéna: *Plovoucí strom prefixů*

Tato scéna je přímou analogií scény s plovoucím vzorem u Knuth-Morris-Prattova algoritmu. Je zde nakreslen text s kurzorem, který je čten a vybarvován jak již bylo vysvětleno dříve. Nad textem je nakreslen strom prefixů, ale bez kořene a tak, že větve nepostupují shora dolů ale zleva doprava zhuštěným způsobem. Strom prefixů klouže ve vodorovném směru tak, že nejdelší prefix některého z vzorů (tedy nejdelší cesta od neznázorněného kořene do některého vrcholu stromu), který je zároveň sufixem přečteného textu, je nastaven tak, aby byl přesně nad zmíněným sufixem. Tato shoda je ve stromu znázorněna zvýrazněním cesty odpovídající prefixu a v textu zvýrazněním odpovídajícího sufixu. Cesta ve stromu prefixů je zvýrazněna bílou barvou, její poslední vrchol budeme nazývat *stav výpočtu*. Na základě výše učiněné úmluvy o znásilnění názvosloví lze tedy stav výpočtu přímo chápat jako prefix.

Bílá barva pro zvýraznění je volena pro zdůraznění analogie se scénami o Knuth-Morris-Prattově algoritmu. U tohoto algoritmu bylo stavem výpočtu číslo. Je vám jistě zřejmé, že to bylo číslo, které říkalo jak daleko je vrchol, který je stavem výpočtu podle algoritmu Aho-Corasick, od kořene. V případě jediného vzoru (kdy se strom prefixů redukuje na cestu) je toto číslo dostatečné pro jednoznačné určení vrcholu, představujícího stav výpočtu jak byl definován v této scéně, pokud je ale vzorů více, může být více i vrcholů v dané vzdálenosti od kořene a pro jednoznačné určení stavu výpočtu již číslo nestačí.

Podobně jako dříve vede způsob určování polohy plovoucího stromu k tomu, že v dokončovací fázi je strom fixován s textem a posouvá se o 1 pole vlevo (pokud maximální shoda prefix-sufix se rozšiřuje o další znak) nebo sklouzne vpravo (popřípadě zůstává stát, zatímco se text pohybuje). Nyní je jistě zřejmé, že strom prefixů byl konstruován tak, aby se sufixem přečteného textu se mohla shodovat nejvýše jedna nad ním ležící cesta ve stromu a začínající v kořeni.

Strom prefixů i tabulka vzorů jsou v dolní části pro informaci znázorněny také a je v nich také ukázáno zvýraznění prefixu a odpovídající cesty s jedinou výjimkou: vrchol stromu představující stav cesty není úplně bílý, ale slabě růžový.

Pokud stav cesty (narůžovělý vrchol), má neprázdnou informaci OUT, pak byl nalezen výskyt vzoru v textu. Jedná se o vzor (nebo vzory) s pořadovým číslem přímo daným informací OUT narůžovělého vrcholu, představujícího stav výpočtu.

Povšimněte si, že pokud dojde ke sklouznutí stromu prefixů vpravo, nová zvýrazněná cesta ve stromu *nemusí* být úsekem předchozí cesty shodné s textem, ale může vybíhat do zcela jiné větve stromu.

Scéna: *Funkce F*

Netriviální je v předchozí scéně pouze jedno: jak určit novou polohu plovoucího stromu prefixů, pokud se aktuální shoda prefixu vzoru a sufixu přečteného textu nedá rozšířit o čtený znak. Z výkladu o algoritmu Knuth-Morris-Pratt už ale řešení v podstatě víte.

Uvažujme jistý okamžik vyhledávání. Stav výpočtu v následujícím okamžiku, po přečtení dalšího znaku textu, bude nejdelší prefix, který vznikne jako prodloužení některého prefixu, který je v tomto okamžiku shodný s textem pod ním, o znak textu, který bude přečten. Může se ovšem stát, že prodloužení žádného ze současných shodných prefixů není prefixem; v takovém případě bude následujícím stavem prázdný prefix, neboli kořen stromu.

Stačí tedy probírat prefixy, které jsou v současné době shodné, monotónně podle délek od nejdelšího (tedy stavu výpočtu) až po nejkratší (kterým je vždy prázdné slovo, tedy slovo o 0 znacích) a jakmile narazíme na první, který jde prodloužit o čtený znak na prefix některého vzoru, pak jeho prodloužení o čtený znak je budoucí stav výpočtu. (Pokud na žádný nenarazíme, bude příští stav výpočtu prázdný řetězec, neboli kořen stromu prefixů.)

Víme již také, že pokud je jistý prefix v jistý okamžik shodný s textem pod ním, pak lze bez znalosti textu určit v etapě předběžného zpracování, které kratší prefixy jsou shodné, protože shodnost prefixu se sufixem textu nám požadovanou informaci o potřebné části textu dá. Pro každý prefix s si tedy jako $F(s)$ označíme nejdelší z prefixů kratších než s, který je shodný, pokud je shodný prefix s. Pak prefixy, jejichž prodlužitelnost zkoumáme, budeme probírat v následujícím pořadí:

$$s \leftarrow \text{stav výpočtu};\ \textbf{while}\ s \text{ není nulový } \textbf{do}\ s \leftarrow F(s);$$

Funkce G, která byla zavedena v popisu scény o vytváření stromu prefixů, má tu vlastnost, že $G(s,a)$ je definována právě když prefix s, prodloužený o znak a, je zase prefix některého vzoru a hodnota $G(s,a)$ je prefix, který je tímto prodloužením. Pak lze změnu stavu s výpočtu popsat následujícím způsobem, kde a označuje nově přečtený znak textu:

$s \leftarrow$ stav výpočtu;
while s není nulový **and** $G(s,a)$ není definováno **do** $s \leftarrow F(s)$;
if $G(s,a)$ je definováno **then** $s \leftarrow G(s,a)$;

Jak se bude funkce F počítat, si ukážeme v příští scéně; v této ji spočítala Algovize, ale ukážeme si její použití. Poznamenávám jen, že označení funkce pochází z anglického "Failure" neboli "Selhání" - míní se akce, která se má provést, selže-li pokus o prodloužení prefixu shodného s textem o další znak. Označení funkce G zase pochází od "Goto" neboli "Jdi do", neboť označuje vrchol do kterého se přejde s označením stavu výpočtu, pokud rozšíření shody je možné.

Funkce F je zapsána ve stromu prefixů v pravé spodní části obrazovky. Každý vrchol stromu nyní má čtyři pole. Levé z nich je jeho pořadové číslo, které opět slouží pouze pro jeho jednoznačnou identifikaci. Další je znakové pole, které bylo používáno ve všech předchozích scénách a napravo od něj je pole, které udává pro vrchol s hodnotu funkce $F(s)$ pro tento vrchol (přesněji řečeno pořadové číslo vrcholu $F(s)$). Vpravo je pole OUT, které již znáte.

Pro názornost je F znázorněna i interaktivně: klepněte na libovolný vrchol do jeho pole udávajícího funkci F. Zvolený vrchol s zežloutne a současně zrůžoví vrchol $F(s)$. Klepnete-li ale do pole pravým knoflíkem myši, zrůžoví nejen $F(s)$, ale i další vrcholy $F(F(s))$, $FF(F((s)))$, atd. tedy další prefixy, které probírá algoritmus při určování následujícího stavu výpočtu. Probírá je pochopitelně od nejdelšího k nejkratšímu, tedy zdola nahoru.

Povšimněme si, že u kořene není hodnota F uvedena. Naše definice se na kořen nedá aplikovat, ale z kódu uvedeného výše je vidět, že hodnotu F ani nepotřebujeme a proto se jí nebudeme zabývat.

Scéna: *Vyhledávání*

Nyní si již zkusíme vyhledávání s využitím stromu prefixů a funkce F. Začínáme ve stavu daném kořenem stromu stavů a podobně jako u Knuth-Morris-Prattova algoritmu pro každý přečtený symbol testu provádíme následující akci:

nejprve postupujeme ve stromu stavů z aktuálního stavu směrem ke kořenu, používajíce funkci F tak dlouho, dokud nenarazíme na stav s', který buď má následníka označeného právě čteným symbolem textu (tedy takový, že $G(s', a)$ je definováno, kde a je čtený symbol) a nebo je roven kořenu stromu.
V prvním případě pak stav výpočtu změníme na zmíněného následníka vrcholu s', tedy na $G(s', a)$, v druhém případě stav změníme na kořen stromu.

Někdy se stane, že výše uvedený stav s' splňuje obě podmínky, tedy je kořenem, ale $G(\text{kořen}, \text{čtený symbol})$ je definováno. Pak budeme postupovat podle prvního případu a stav změníme na prefix délky 1 určený čteným symbolem.

Pokud se při výpočtu po provedení dokončovací fáze dostaneme do stavu s, pro který je $OUT(s)$ neprázdné, pak algoritmus ohlásí nalezení výskytu vzoru nebo vzorů, uvedených v poli $OUT(s)$.

V této scéně jsou hodnoty G, F a OUT všech vrcholů vypočteny Algovizí a uvedeny ve stromu prefixů, abyste si mohli zkusit vyhledávání podle algoritmu Aho-Corasick a ověřit si vlastnosti funkcí. F a OUT jsou uvedeny explicitně ve třetím a čtvrtém poli vrcholu způsobem popsaným v předchozí scéně, G se určí prohlédnutím následníků uvažovaného vrcholu a jejich označení symboly abecedy.

Povšimněte si také, že ze stejných důvodů, jako tomu bylo u Knuth-Morris-Prattova algoritmu, je počet skoků na kratší prefix ve čtecí fázi podle funkce

F za celou dobu výpočtu nejvýše roven počtu znaků textu: délka aktuálního prefixu ukazovaného šipkou se prodlouží o 1 v dokončovací fázi a těchto fází je tolik, kolik je znaků textu. Při přechodu na kratší prefix se jeho délka zkrátí o alespoň 1; přitom výpočet začíná prefixem délky 0 a jeho délka nikdy nemůže být záporná a proto počet zkrácení je nejvýše roven počtu prodloužení.

Scéna: *Určování funkce F*

Výpočet funkce F budeme provádět postupně podle délky prefixu, tedy nejdříve pro prefixy délky 1 (což jsou následníci kořene - pro kořen se hodnota funkce F neurčuje), pak pro prefixy délky 2, atd. Výhodou tohoto postupu je, že pokud určujeme $F(s)$, známe již hodnoty F pro všechny prefixy kratší, což v dalším využijeme. Pro zjednodušení vrcholy zobrazujeme bez pole udávajícího hodnoty OUT.

Knoflíkem **Krok** určíme hodnotu F prefixu najednou, je pro zpracovávaný prefix znázorněna červenou šipkou ve stromu prefixů a doplněna ve třetím poli příslušného vrcholu. Projděte si takto výpočet pro několik krátkých prefixů, kde výpočet je triviální a pak začněte výpočet krokovat detailně pomocí knoflíku **Fáze.**

Krok: *Výchozí stav*

Po prvním klepnutí na knoflík **Fáze** se zobrazí výchozí situace: představte si, že s je stavem výpočtu a tedy se shoduje s koncovým úsekem (sufixem) přečteného textu. Strom prefixů je vyobrazen dvakrát, jednou v dolní části okna a jednou jako plovoucí nad textem. Jelikož se výpočet provádí v etapě předzpracování, text není znám, ale ve stromu nad textem je zvýrazněna cesta, odpovídající zpracovávanému prefixu s, ve spodním stromu je jen bílou barvou ukázán vrchol odpovídající prefixu s. Plovoucí strom je ukázán v poloze, ve které je zvýrazněný prefix nad pravým koncem dosud přečteného textu. ◇

Krok: *Určení textu určeného stavem*

Další klepnutí na knoflík **Fáze** ukáže některé symboly v textu. Jsou to ty, které se musí shodovat s odpovídajícími symboly zvýrazněného prefixu ve stromu. ◇

Krok: *Vrácení dokončovací fáze*

Do znázorněné situace bychom se při vyhledávání řetězce dostali dokončovací fází předchozího kroku. Klepnutím na **Fáze** se posuneme před tuto fázi. Stavem výpočtu tehdy byl prefix s' řetězce s, který se dostane odtržením jeho posledního symbolu. Tento odtržený symbol, který po dokončovací fázi byl posledním symbolem přečteného textu, se přitom stane po vrácení dokončovací fáze čteným symbolem textu.

Ve stromu v dolní části okna se objeví růžová šipka, ukazující od výchozího prefixu s, jehož hodnotu F určujeme, k prefixu s', který je stavem před dokončovací fází, který je ukázán nahoře v plovoucím stromu. ◇

Krok: *Prefixy zkráceného stavu*

Další aplikace knoflíku **Fáze** nechávají sklouzávat plovoucí strom do kratších prefixů, které jsou také shodné se sufixem přečteného textu v okamžiku před dokončovací fází. Provádíme to iterovanou aplikací funkce F na prefix s'. Ve spodním stromu se aplikace funkce F znázorňuje modrými šipkami. ◇

Krok: *Opětovné provedení dokončovací fáze*

Postup v předchozím kroku opakujeme tak dlouho, dokud nenarazíme na stav t, pro který lze shodu s textem přečteným před dokončovací fází prodloužit na čtený symbol. Dojde-li k tomu, pak provedeme knoflíkem **Fáze** zpět dokončovací fázi; prefix t spolu s čteným symbolem vytvoří prefix shodný se sufixem přečteného textu a je zřejmé, že to je nejdelší vlastní prefix výchozího stavu s, který je také sufixem přečteného textu, neboli je roven hodnotě $F(s)$.

V dolním stromu je tento přechod znázorněn žlutou šipkou.

Pokud žádný (ani prázdný) prefix řetězce s' nelze rozšířit o symbol čtený před dokončovací fází, pak hodnota $F(s)$ bude rovna kořeni stromu. ◇

Zatímco pomocí plovoucího stromu jsme ukázali logické zdůvodnění výpočtu funkce F, šipky v dolním stromu ukazují postup aplikace operací ve stromu, které k hodnotě $F(s)$ vedou.

Část VII

Lineární programování

Lineární programování je odvětví spojité optimalizace, kde se hledá největší nebo nejmenší možná hodnota lineární funkce (funkcionálu) na konvexním polyedru v n-dimenzionálním Euklidovském prostoru. Lineární programování je důležité jednak samo o sobě jako užitečná a velmi často používaná metoda a také pomocí něho je možno přeformulovat mnoho úloh, které s lineárním programováním nemají zdánlivě nic společného, například hledání největšího toku v síti.

Z celé bohaté teorie i algoritmiky lineárního programování uvedeme jediné - ilustraci simplexového algoritmu v rovině nebo 3-dimenzionálním prostoru. Naším cílem není přesný a detailní popis algoritmu, ale spíš vybudování geometrické intuice, která nahlíží na simplexový algoritmus jako procházku po polyedru směrem vzhůru, prováděnou přesuny po hranách polyedru.

Kapitola 25

Simplexový algoritmus

V úvodu této kapitoly je nutno uvést, že tento applet ukazuje pouze jednu partii, která je v teorii lineárních programů důležitá. Vůbec se například nebudu zabývat dualitou v lineárním programování, což je zdaleka nejdůležitější součást celé teorie, protože je to svojí podstatou matematický vztah, který neumím výstižně vizualizovat. Z tohoto důvodu také nebudou probírány další důležité pojmy primární a duální úlohy a z nich vyplývající metody výpočtu. Zcela mimo zůstanou i otázky celočíselnosti řešení a aplikace lineárního programování a také algoritmy pracující zaručeně v polynomiálním čase, jako je například elipsoidová metoda a metody vnitřního bodu. Zaměřím se zde na výklad základních myšlenek simplexového algoritmu.

Při definici problému i popisu algoritmu se nebudu striktně držet obvyklého formalismu, například výkladu t. zv. standardní formy úlohy lineárního programování, což je výhodné například při odvozování duality, ale zbytečně zatěžuje čtenáře obraty, které nejsou důležité pro pochopení myšlenky simplexového algoritmu. Lineární programování budu chápat jako lineární optimalizaci nad obecným polyedrem a simplexový algoritmus jako putování mezi vrcholy polyedru, které je monotónní vzhledem k optimalizovanému funkcionálu.

V této kapitole se nejprve budeme zabývat optimalizací v rovině a později v třídimenzionálním prostoru, neboť tyto případy je možno snadno vizualizovat a vybudujeme tak geometrický přístup k LP a simplexovému algoritmu konkrétně, který pak je snadné zobecnit na vícedimenzionální případ.

Scéna: *Přímky*

Body roviny, vyhovující rovnici $ax + by = c$, kde a a b nejsou současně rovny nule (a x a y představují x-ovou a y-ovou souřadnici uvažovaného bodu) vytvářejí přímku a naopak každá přímka se takto dá popsat.

V této scéně je znázorněna souřadná soustava roviny s osami x a y opatřenými měřítkem. Nakreslete přímku tak, že v jistém místě obrazovky stisknete myš a v jiném místě ji uvolníte. Algovize body spojí přímkou a připíše k ní její rovnici. Než bude knoflík myši uvolněn, je přímka kreslena červenou barvou, místo kde byla myš stisknuta je označeno malým kolečkem a okamžitá poloha kurzoru je označena šipkou. V okamžiku uvolnění myši kolečko a šipka zmizí a přímka je zafixována a kreslena tmavě modře. Kreslení přímky lze vícenásobně opakovat. Současně s nakreslením přímky se na straně obrazovky objeví pole, ve kterém je napsána její rovnice (pro právě kreslenou přímku žluté a pro nakreslené přímky bílé).

V kontrolním panelu je ale možno zvolit i další módy funkce myši. Kromě již popsané volby **Kresli přímku** lze také vybrat volbu **Zvol a edituj přímku, Vynech přímku** a **Posuň nerovnosti.**

Při volbě **Zvol a edituj přímku** se některá z nakreslených přímek zvolí tím, že se na ni klepne myší. Zvolená přímka ze zobrazí fialově s kolečkem a šipkou, jak na ní byly v posledním okamžiku, kdy byla kreslena. Současně se barevně odliší její rovnice (žluté pole). Někdy tuto funkci volíme jen proto, abychom si přiřadili zvolené přímce její rovnici; jindy ale použijeme možnosti přímku změnit tím, že tažením myší přemístíme její kolečko nebo šipku. Pokud klepneme jinam než na zvolenou přímku, volba přímky se anuluje (a bylo-li klepnuto na jinou přímku, tato je zvolena a může vzápětí být editována).

Při volbě **Vynech přímku** se klepnutím na přímku tato přímka vynechá. Při této volbě se také objeví jinak skrytý knoflík **Vynech vše,** kterým je možno vynechat všechny nakreslené přímky a popřípadě (po změně volby) znovu začít přímky kreslit.

Při volbě **Posuň nerovnosti** je možno myší přetáhnout blok nerovností do jiné polohy, nic jiného se myší nedá dělat. Knoflíky + a **-** se dá velikost okénka s nerovnostmi měnit a vypnutím **Ukaž nerovnosti** je možné zobrazování nerovností vypnout. Zřejmým způsobem lze i skrýt osy souřadnic.

Poznamenejme, že tutéž přímku lze popsat mnoha rovnicemi. Jestliže všechny tři parametry a, b a c v rovnici přímky vynásobíme týmž nenulovým číslem, dostaneme jiný popis téže přímky (a naopak lze od jednoho popisu k druhému tímto způsobem vždy přejít). Popis uváděný na obrazovce je volen tak, aby platilo $c = 1000$, čímž je rovnice přímky jednoznačně určena. (Čísla a a b jsou v zápisu na obrazovce zaokrouhlena.)

Scéna: *Poloroviny*

Přímka rozetne rovinu na dvě části - *poloroviny.* Polorovinu chápeme tak, že zahrnuje i hraniční přímku (je to tedy *uzavřená* polorovina). Průsečíkem obou polorovin je přímka, která je určuje.

Poloroviny určené přímkou s rovnicí $ax + by = c$ je možno popsat jako množiny bodů $[x, y]$, které vyhovují nerovnosti $ax+by \leq c$ respektive $ax+by \geq$

c. Druhou nerovnost je možné psát také jako $(-a)x + (-b)y \leq -c$ a proto vhodnou volbou kladných a záporných koeficientů je možno popsat každou polorovinu nerovností se znaménkem $\leq$ a konstantou na pravé straně a tím sjednotit zápis.

Nakreslete stejným způsobem jako v předchozí scéně přímku. Rozdělení na poloroviny se barevně zvýrazní. (Zopakujme, že každá z polorovin zahrnuje i hraniční přímku, jak je z rovnic ostatně vidět). V dalším budeme vždy uvažovat polorovinu znázorněnou zeleně. Místo šipky v této scéně používáme pro znázornění orientace přímky položipku, která leží celá v uvažované zelené polorovině.

Pokud není knoflík myši stisknut, v zobrazení rovnice je uvedena i hodnota levé strany pro bod, daný okamžitou polohou kurzoru. Barevně je také rozlišeno, zda nerovnost platí. Je-li splněna jako ostrá nerovnost, je okénko nerovnosti světle zelené, platí-li rovnost, je tmavě zelené. Pokud nerovnost neplatí, je okénko fialové. Ověřte si, že okénko je (světle nebo tmavě) zelené, pokud kurzor ukazuje do zvolené poloroviny a je fialové, je-li mimo ni.

Na rozdíl od předchozího případu se nakreslením nové přímky předchozí přímka i rozdělení roviny jí určené vymaže.

Scéna: *Lineární nerovnosti a konvexní mnohoúhelník*

V této kapitole se budeme zabývat konvexními mnohoúhelníky. Konvexní mnohoúhelník v rovině je množina bodů $[x, y]$, které vyhovují soustavě lineárních nerovností

$$a_1 x + b_1 y \leq c_1$$

$$a_2 x + b_2 y \leq c_2$$

$$\dots$$

$$a_m x + b_m y \leq c_m$$

kde $a_1, b_1, c_1, \dots, a_m, b_m, c_m$ jsou konstanty. Jelikož množina bodů, které vyhovují jedné z těchto nerovností, je polorovina, je konvexní mnohoúhelník v rovině průnikem polorovin.

V této scéně je možno konvexní mnohoúhelník budovat postupným přidáváním lineárních nerovností. Přímka se kreslí stejně jako v předchozí scéně, ale nakreslením nové přímky dříve nakreslené přímky nezmizí, ale zůstávají na obrazovce. Je možno je editovat jako tomu bylo v první scéně.

Vidíme tedy současně mnoho zvolených polorovin. Body roviny jsou znázorněny takto: Ty, které leží ve všech polorovinách a tedy představují konvexní mnohoúhelník, daný jejich průnikem, jsou zelené. Zelená oblast je tedy mnohoúhelník definovaný nerovnostmi v rámečku. Ostatní body jsou modré, ale je odlišeno, v kolika polorovinách leží. Body, které leží v hodně polorovinách (ale ne ve všech), jsou tmavě modré, tím tmavší, čím větší počet polorovin

je obsahuje. Naopak body, ležící v málo polorovinách jsou světlé a ty, které nejsou v žádné (možná ale takové body v danou chvíli nejsou) jsou takřka bílé. Pokud není stisknut knoflík myši, pak pro bod ukazovaný kurzorem jsou také barevně odlišeny nerovnosti stejným způsobem jako v minulé scéně. Pohybujte kurzorem a sledujte, které nerovnosti jsou splněny a které ne.

Knoflíkem **Nový polyedr** je také možno vytvořit náhodně nový polyedr, který je pak možno upravovat přidáváním, ubíráním nebo přesouváním hraničních přímek. Checkboxem **Pouze polyedr** se přepne do módu, kdy všechny body mimo zelený mnohoúhelník mají stejnou barvu. Není pak rozlišeno v kolika polorovinách jsou obsaženy, ale jen zda patří do všech nebo ne. Tato volba je užitečná pokud nás zajímá jen samotný mnohoúhelník polorovinami vytvořený.

Mnohoúhelník může být neomezený, omezený a přitom neprázdný a nebo prázdný. Pro jednu nerovnost ($m = 1$) je tvořen neomezenou polorovinou, pro dvě nerovnosti je obecně představován neomezeným klínem (ale pro rovnoběžné hraniční přímky může být tvořen polorovinou, nekonečným pásem konstantní šířky, přímkou nebo být prázdný). Zkuste nakreslit všechny tyto možnosti.

Počínaje třemi nerovnostmi mnohoúhelník může (ale nemusí) být omezený nebo i prázdný i v případě nerovnoběžných hraničních přímek.

Přidáním další nerovnosti k soustavě, která má alespoň jedno řešení (neprázdný mnohoúhelník) se stane jedna z následujících možností

1. mnohoúhelník se zmenší, ale zůstane neprázdný - to znamená, že některé body původního mnohoúhelníku nové nerovnosti vyhovují a některé ne,

2. mnohoúhelník se nezmění - všechny body mnohoúhelníku nové rovnici vyhovují, což znamená, že platnost nové nerovnosti lze odvodit z nerovností předcházejících,

3. nové nerovnosti nevyhovuje žádný bod stávajícího mnohoúhelníku - mnohoúhelník se stane prázdným, nová nerovnost je ve sporu s předcházejícími.

Scéna: *Lineární funkcionál*

Lineární funkcionál je funkce, která každému bodu $[x, y]$ roviny přiřadí číslo $Ax + By$, kde A a B jsou konstanty. Na obrazovce jsou graficky znázorněny hodnoty funkcionálu s náhodně volenými hodnotami A a B. Hodnoty jsou znázorněny pomocí barev tepelné škály: nejvyšší hodnoty, které je vidět na obrazovce, jsou znázorněny bíle, přecházejí do stále temnější žluté, pak červené, z nejtmavší červené do tmavnoucí modré, až se nakonec stanou černými (ne všechny barvy musí být na obrazovce viditelné). Je také nakreslena

souřadná soustava s měřítky a šipka představující vektor (A, B), který začíná v počátku a vede do bodu $[A, B]$.

Budete-li pohybovat kurzorem se stisknutým knoflíkem myši, konec vektoru bude vždy v místě kurzoru. Výjimkou je, pokud je kurzor příliš blízko počátku (bráním se zobrazování funkcionálu daného nulovým vektorem) a nebo pokud je příliš daleko (barevné pole by bylo úzké a na okraji okna by se nedostávaly barvy).

Je vidět, že hodnoty funkcionálu se nejrychleji mění ve směru vektoru (A, B), zatímco ve směru na něj kolmém se nemění. Je to dáno tím, že hodnota funkcionálu v bodě $[x, y]$ je dána skalárním součinem vektoru (A, B) s vektorem (x, y), začínajícím v počátku a končícím v bodě $[x, y]$. Skalární součin vektoru (A, B) s přírůstkem, který je na něj kolmý, je nulový, zatímco největší je pokud jsou oba vektory kolineární.

Na obrazovce jsou také znázorněny některé "vrstevnice", čáry spojující body mající stejnou hodnotu funkcionálu. Jsou tvořeny šedě nakreslenými přímkami, které jsou kolmé na vektor určující směr nejrychlejšího přírůstku funkcionálu a jsou tedy navzájem rovnoběžné.

Je zřejmé, že čím je vektor $[A, B]$ delší, tím rychleji funkcionál v jeho směru roste, naopak pro velmi krátký vektor je funkcionál v pozorovatelné oblasti skoro konstantní.

Jak uvidíte dále, pro naše účely nebude důležité, jak rychle funkcionál roste, ale ve kterém směru roste nejrychleji. Jinými slovy, bude nás zajímat pouze směr, ale nikoli délka vektoru (A, B), který určuje funkcionál.

V rohu obrazovky je malé okénko, které ukazuje hodnotu funkcionálu v místě ukazovaném kurzorem. Se stisknutým knoflíkem myši zkuste měnit směr vektoru, určujícího funkcionál a pak s uvolněným knoflíkem sledujte změnu hodnoty funkcionálu při pohybu kurzorem.

Scéna: *Lineární optimalizace v rovině*

V této scéně se dostáváme k základní úloze, kterou budeme v rovině řešit. Na obrazovce je nyní zobrazen konvexní mnohoúhelník a současně lineární funkcionál a máme nalézt ten bod mnohoúhelníku, pro který je hodnota funkcionálu největší.

Mnohoúhelník na obrazovce je vygenerován náhodně a zobrazen zeleně. Jako podklad pod ním je provedeno grafické znázornění funkcionálu využitím tepelné škály a vrstevnic, jak byly vysvětleny v předchozí scéně.

Řešením problému lineární optimalizace je ten bod mnohoúhelníku, který leží na nejsvětlejším možném místě funkcionálu. Bod, který je řešením, je znázorněn malým červeným kolečkem. Pohybem kurzoru při stisknutém knoflíku myši měňte funkcionál a sledujte řešení.

Jistě je vám zřejmé, že řešením je vždy některý vrchol mnohoúhelníku, tedy místo, kde se hranice mnohoúhelníku zalamuje. Jen občas dojde k výjimce,

kdy množinou řešení je celá jedna hrana mnohoúhelníku, spojující dva jeho sousedící vrcholy.

Pro změnu zadání stiskněte knoflíku **Nový polyedr,** ukáže se jiný konvexní mnohoúhelník a funkcionál a opět je možno zobrazit si řešení. Volbou na panelu ovladače lze vybrat, zda má mnohoúhelník být omezený nebo neomezený. Pro zjednodušení scény není implementována možnost editovat mnohoúhelník.

Jistě vám je jasné, že pokud je zelený konvexní mnohoúhelník neprázdný a omezený, má úloha lineární optimalizace vždy řešení. Pokud ale je mnohoúhelník neomezený, řešení může existovat, ale také nemusí. Druhý případ nastává, pokud vektor určující funkcionál ukazuje směrem, který svírá ostrý úhel s alespoň jednou polopřímkou, která leží celá v neomezeném mnohoúhelníku, protože podle takové polopřímky rostou hodnoty funkcionálu do nekonečna, takže konečné maximum hodnot funkcionálu v bodech mnohoúhelníka neexistuje.

Zvolte si nyní možnost **Neomezený polyedr** a sledujte jak existence řešení závisí na směru růstu lineárního funkcionálu.

Úloha nalézt řešení lineárního programu v rovině, tedy se dvěma proměnnými x a y, není složitá a i pro velký počet nerovností by nebylo těžké ji vyřešit i jinými způsoby, než je metoda popisovaná v dalších scénách, ale postup, který ukážeme, je snadné zobecnit pro libovolný počet proměnných a pod názvem "simplexový algoritmus" je i v současné době patrně nejpoužívanější metodou pro řešení lineárních programů i přes to, že byla popsána G. B. Danzingem v již roce 1947.

Scéna: *Pohyb po hraně v rovině*

Jak jsme viděli v předchozí scéně, pokud má úloha lineárního programování řešení, tak toto řešení leží na obvodu mnohoúhelníka daného nerovnostmi a je jím vrchol tohoto mnohoúhelníka nebo, pokud je řešení více, pak řešeními jsou dva sousedící vrcholy mnohoúhelníka a všechny body hrany mezi nimi. Simplexový algoritmus hledá řešení tak, že začne v libovolném vrcholu mnohoúhelníka daného nerovnostmi a pohybuje se po obvodu mnohoúhelníka, dokud se nedostane do vrcholu, který je řešením (respektive jedním z řešení).

Podívejme se nyní na to, jak probíhá přechod z jednoho vrcholu mnohoúhelníku do vrcholu sousedního. Poznamenávám, že v této scéně je mnohoúhelník vždy neprázdný a omezený. Na obrazovce se objevil žlutý žeton, jehož střed bude stále na obvodu mnohoúhelníka. Kotoučkem je možno pohybovat tak, že klepnete na knoflík **Jdi vlevo** nebo **Jdi vpravo** a žeton se dá do pohybu po obvodu mnohoúhelníka do sousedního vrcholu ve zvoleném směru. "Jít vpravo" znamená ve směru hodinových ručiček, protože tak je nutné točit volantem, chceme-li zatočit vpravo, podobně vlevo je proti směru hodinových ručiček. Chvíli si pohyb po obvodu zkoušejte.

Nebaví-li vás znázorněný mnohoúhelník, můžete si nechat zkonstruovat jiný knoflíkem **Nový polyedr.** Nejspíš vás ale nebaví tato scéna, vždyť je to tak jednoduché. Podíváme se ale ještě, jak určit souřadnice souseda daného vrcholu ve směru dané hrany, když se nedíváme na obrázek, ale chystáme se vysvětlovaný algoritmus programovat a máme k dispozici jen nerovnosti určující mnohoúhelník a souřadnice vrcholu s žetonem.

Výklad ve zbytku scény můžete přeskočit, pokud Vám jde jen o pochopení myšlenky simplexového algoritmu. Jestliže však budete chtít algoritmus také programovat, dobře si výklad promyslete.

Místo knoflíků **Jdi vlevo/vpravo** stiskněte jeden z knoflíků **Vysvětli vlevo** nebo **Vysvětli vpravo.** Místo aby se žeton přesunul do jednoho ze sousedních vrcholů se na obrazovce nakreslí vše potřebné pro ilustraci následujícího výkladu, který si klade za cíl zjistit vrchol do kterého by se žeton při stisknutí knoflíku přesunul:

- Na obrazovce se objevily všechny přímky, které procházejí hranami mnohoúhelníka. Jsou to přímky, jejich rovnice dostaneme tak, že v nerovnostech určujících mnohoúhelník nahradíme symbol $\leq$ symbolem $=$ (rovnost).

- Z vrcholu, ve kterém je žeton, vycházejí dvě hrany. Ta po které by měl přejít žeton po stisku knoflíku **Jdi ...** a která je nyní nakreslena jako červená šipka a je součástí přímky nakreslené červeně, se bude nazývat *pivotní*, druhá hrana je *nepivotní*.

- červenými a černými kolečky jsou označeny průsečíky červené přímky, procházející pivotní hranou, s tmavomodrými přímkami procházejícími ostatními hranami (některé průsečíky možná leží mimo obrazovku).

- Jak jste si jistě všimli, polorovina určená nerovností, která odpovídá nepivotní hraně, se zbarvila tmavěji než je původní barva pozadí (ale je částečně zakryta mnohoúhelníkem). Průsečíky, zmíněné v předchozím odstavci, jsou kresleny červeně, pokud leží v tmavě modré polorovině dané nepivotní hranou, a jsou kresleny jako černé v opačném případě.

- Vrchol pivotní hrany, do kterého by se žeton přesunul, je ten z bodů označených červeným kolečkem, který nejbližší vrcholu se žlutým žetonem.

Určení průsečíku přímek procházejících dvěma vybranými hranami je přitom jednoduché: hranám odpovídají nerovnosti ze souboru, určujícího mnohoúhelník. Nahradíme-li v nerovnostech $\leq$ symbolem $=$, dostaneme soustavu dvou lineárních rovnic o dvou neznámých a jejich vyřešením se získají souřadnice průsečíku.

Určení, které průsečíky leží v polorovině dané nerovností odpovídající nepivotní hraně, je ještě jednodušší: souřadnice průsečíku se dosadí do příslušné nerovnosti a zjistí se, zda je splněna.

Scéna: *Pohyb po hraně neomezeného mnohoúhelníka*

Scéna je opakováním předchozí, ale pro neomezený mnohoúhelník. Některé jeho hrany jsou neomezené polopřímky. Pustí-li se po takové hraně žeton, nikdy nedorazí do sousedního vrcholu, ale navždy by běžel po polopřímce. Musíte jej proto dostat zpět stisknutím knoflíku pro opačný směr.

I zde je možno přepnou do módu **Výklad;** uvidíte, že pokud by žeton měl běžet do nekonečna po hraně tvořené polopřímkou, pak nevznikne žádný červeně zvýrazněný průsečík, neboť polopřímka tvořící hranu (což je průsečík červené přímky a tmavé poloroviny určené nerovností odpovídající nepivotní hraně) se neprotíná z žádnou z přímek procházejících ostatními hranami.

Scéna: *Simplexový algoritmus v rovině*

Scéna je do jisté míry opakováním předchozích dvou - ukazuje konvexní mnohoúhelník a žeton, který se pohybuje po jeho obvodu; zde jím ale pohybuje program. Navíc je zde ale zobrazen i lineární funkcionál a program žetonem pohybuje tak, aby vždy mířil k vyšším hodnotám funkcionálu.

Je-li na ovladači volba **Funkcionál,** lze myší měnit lineární funkcionál, při volbě **Polož žeton** lze nastavit výchozí polohu žlutého žetonu klepnutím na některý vrchol mnohoúhelníka.

Po nastavení výchozí situace klepněte na knoflík **Počítej.** Algovize určí optimum funkcionálu na mnohoúhelníku pomocí simplexového algoritmu a přepne do animačního módu, kde lze výpočet krokovat nebo nechat volně běžet. Každý krok je představován přechodem žetonu po hraně mnohoúhelníka a kroky se provádějí tak dlouho, dokud není nalezeno optimální řešení problému; pak se žeton zastaví a zčervená. Knoflíkem **Zruš** se vrátíte do módu nastavení a můžete zvolit jinou výchozí situaci nebo přejít do další scény.

Jelikož optimalizujeme v rovině, z každého vrcholu mnohoúhelníka vycházejí dvě hrany. Nejčastěji v průběhu výpočtu žeton leží ve vrcholu, ze kterého jedna hrana směřuje k vyšším hodnotám funkcionálu a druhá (pokud nejsme na začátku výpočtu, je to ta, po které jsme do vrcholu přišli), jde k nižším hodnotám. Pak je jasné, kam žeton posouvat. Na začátku výpočtu někdy mohou oba směry stoupat k vyšším hodnotám funkcionálu, pak je v zásadě jedno, kudy půjdeme. Jestliže v obou směrech funkcionál klesá (nebo alespoň neroste), je žeton v místě nejvyšší hodnoty funkcionálu, tedy bylo nalezeno optimum a výpočet končí a žeton zčervená pro označení výsledku.

Scéna: *Simplexový algoritmus - neomezený mnohoúhelník*

Předchozí scéna je opakována s tím, že nyní budou všechny vytvořené konvexní mnohoúhelníky neomezené. Pokud se stane, že lineární funkcionál stoupá podél nějaké neomezené polopřímky, pak může nabývat libovolně vysokých hodnot a bod mnohoúhelníka s maximální hodnotou funkcionálu neexistuje.

Scéna: *Simplexuj si sám v rovině*

Scéna je stejná jako předchozí dvě, ale žetonem nepohybuje program, ale uživatel pomocí knoflíků **Jdi vpravo** a **Jdi vlevo** tak, jak to bylo ukázáno v jedné z předchozích scén. Je třeba žetonem pohybovat tak, aby stále směřoval k vyšším hodnotám funkcionálu a pokud to již nejde, tj. je nalezeno maximum funkcionálu, pak je třeba stisknout knoflík **Maximum.**

V této scéně je možné si vygenerovat i neomezený mnohoúhelník. Proto je pro řízení výpočtu přidán ještě knoflík **Neomezený,** který je třeba stisknout v okamžiku, kdy z vrcholu vychází hrana tvořená neomezenou polopřímkou, podle které hodnoty funkcionálu rostou. Kotouček nezčervená, protože jeho poloha nepředstavuje optimální polohu (ta neexistuje). Nemačkejte knoflík, pokud z obrázku vidíte, že hodnoty funkcionálu nejsou shora omezeny, ale žádná polopřímka, podle které funkcionál roste do nekonečna, nevychází z vrcholu, ve kterém se nachází žeton, i když vidíte, že takový vrchol existuje.

Při pokynu jít nesprávným směrem a nebo předčasném ohlášení konce výpočtu knoflíky **Maximum** nebo **Neomezený** Algovize příkaz neprovede a protestuje. Chybová hlášení začínají vykřičníky a jsou následující:

- "Klesající směr" - v případě, že byl dán pokyn jít ve směru, ve kterém funkcionál klesá (bez ohledu na to, zda je hrana omezená úsečka nebo neomezená přímka).

- "Není maximum" - v případě, že byl stisknut knoflík **Maximum** v okamžiku, kdy žeton není v místě maxima funkcionálu na mnohoúhelníku.

- "Neomezený směr" - v případě, že byl dán pokyn jít ve směru, ve kterém funkcionál roste podle neomezené hrany; v takovém případě je správně třeba stisknout knoflík **Neomezený.**

- "Shora omezené" - v případě, že byl stisknut knoflík **Neomezený,** ale hodnoty funkcionálu na všech hranách vycházejících z vrcholu označeného žetonem jsou shora omezené (což nastává když hrany jsou buď omezené úsečky bez ohledu na to, zda na nich funkcionál roste nebo klesá a nebo neomezené polopřímky, na nichž funkcionál směrem od vrcholu s žetonem klesá).

Nastavte si libovolně výchozí situaci a doveďte žeton do vrcholu maximalizujícího funkcionál nebo do vrcholu, ze kterého vychází polopřímka, na které funkcionál není shora omezený. Opakujte výpočet z různých výchozích situací, dokud se vám několikrát za sebou nepodaří vše provést bez jediné chyby (tedy bez hlášení s vykřičníky).

Scéna: *Polyedr v prostoru*

V této scéně zopakujeme vše, co bylo výše řečeno pro dvoudimenzionální Euklidovský prostor, tedy pro rovinu, znovu pro třídimenzionální prostor tak, aby nečinilo potíže zobecnit to pro libovolnou konečnou dimenzi.

Rovnice $ax+by+cz = d$, která je obdobou rovnice přímky v rovině, je rovnice roviny ve 3D prostoru. Znázornit rovinu v prostoru na dvoudimenzionální obrazovce není snadné a proto to dělat nebudeme.

Rovina rozděluje 3D prostor na dva podprostory stejně tak jako přímka rozděluje rovinu na dvě poloroviny. Tyto podprostory (chápány jako uzavřené, tedy zahrnující i hraniční rovinu) jsou v případě roviny s rovnicí $ax+by+cz = d$ dány nerovnostmi $ax + by + cz \leq d$ a $ax + by + cz \geq d$. Druhou budeme v zájmu jednotnosti zápisu psát ve formě $a'x + b'y + c'z \leq d'$, kde $a' = -a$, $b' = -b$, $c' = -c$ a $d' = -d$. Stejně jako rovinu, ani poloprostor nelze na dvojdimenzionální obrazovce snadno znázornit.

Polyedr je průnik konečného množství poloprostorů. Je to konvexní těleso, jehož příklad je znázorněn v této scéně. Matematicky je to tedy množina bodů třídimenzionálního prostoru, která vyhovuje soustavě nerovností

$$\begin{aligned} a_1x + b_1y + c_1z &\leq d_1 \\ a_2x + b_2y + c_2z &\leq d_2 \\ &\dots \\ a_mx + b_my + c_mz &\leq d_m \end{aligned}$$

Polyedr jde ve většině případů na rovinné obrazovce znázornit snadno. Tato scéna nabízí volbu z několika připravených polyedrů. Na ovladači je volba mezi náhodným polyedrem, dokonalými Platonovskými tělesy a několika jehlany. Při volbě **Náhodný** je možno vygenerovat nový náhodně vytvořený polyedr knoflíkem **Nový polyedr,** přitom lze zvolit vytvoření omezeného nebo neomezeného polyedru. V poli **N=** je možné nastavit požadovaný počet nerovností. Skutečný počet stěn polyedru ale bude obvykle jiný. Pokud je hodnota N příliš malá a je požadován *omezený* polyedr, často se stane, že N náhodných nerovností omezený polyedr nevytvoří a proto pak Algovize přidává další náhodné nerovnosti až do dosažení omezenosti. Na druhé straně se stává, že některé nerovnosti jsou zbytečné - jejich vynechání polyedr nezmění, což také znamená, že nevytvářejí žádnou novou stěnu. Takové nerovnosti nejsou ani zobrazovány.

Polyedrem na obrazovce můžeme pohybovat myší: jestliže je na ovladači zvolena volba **Rotuj,** pak horizontální pohyb myši rotuje polyedrem kolem svislé osy a vertikální pohyb myši kolem vodorovné osy ležící v rovině obrazovky. Pokud je zvolena volba **Pohybuj,** pak pohybem myši táhneme polyedr po obrazovce, aniž by se natáčel. Nakonec při volbě **Zoomuj** vertikální pohyb myši polyedr zvětšuje nebo zmenšuje bez natáčení (horizontální pohyb myši nemá na zobrazení vliv).

Poznamenejme, že polyedr může být omezený, ale také neomezený (volba **Neomezený).** Při otáčení neomezeného polyedru se pak může stát, že se díváme (z nekonečna) dovnitř; je tam tma a vidíme jen bíle naznačené hrany. Vnitřní body sice do polyedru patří, ale pro názornější zobrazení je polyedr zobrazen jako dutý, protože simplexový algoritmus pracuje výhradně na jeho povrchu. V případě omezeného polyedru předpokládáme, že pozorovatel je dostatečně vzdálen, takže zůstává stále vně polyedru.

Anatomie 3D polyedru je obdobná jako u konvexního mnohoúhelníka v rovině, ale složitější. Body, které splňují všechny nerovnosti definující polyedr jako ostré, jsou ve vnitřku polyedru a v třídimenzionálním prostoru je pozorovatel mimo polyedr nevidí. Body, které všechny nerovnosti splňují tak, že alespoň v jedné nerovnosti platí rovnost, jsou na povrchu neboli hranici polyedru. Všechny body polyedru, které splňují jistou nerovnost jako rovnost, vytvářejí *stěnu* polyedru. Říkáme, že nerovnost a stěna si navzájem odpovídají. Mnohé pravidelné polyedry (dokonalá neboli Platonova tělesa) se označují nebo pojmenovávají počtem jejich stěn; například krychle je šestistěn, pravidelný trojboký jehlan je čtyřstěn, známe i osmistěn, dvanáctistěn a dvacetistěn. Pravidelná tělesa je také možno si zobrazit příslušnou volbou na ovladači.

Na obrazovce jsou také vypínatelně zobrazeny nerovnosti, které polyedr definují. Barva zde ale neoznačuje, zda je nerovnost splněna, ale nerovnosti odpovídající viditelným stěnám mají podklad stejné barvy jsou má odpovídající stěna.

Nerovnosti, které odpovídají stěnám, které pro dané natočení polyedru nejsou viditelné, jsou napsány šedě na černém pozadí. Dále v seznamu mohou být nerovnosti, které by bylo možno vynechat, aniž by se polyedr změnil. Jinými slovy jsou to takové nerovnosti, kterým vyhovují všechny body aktuálního polyedru. Tyto nerovnosti jsou zobrazeny na světle šedém pozadí bílým písmem.

V této scéně (a jenom v ní) jsou vlevo vedle nerovností malé bílé čtverečky. Klepnutím na čtvereček se nerovnost inaktivuje: čtvereček ztmavne (označuje neaktivní nerovnost) a příslušná nerovnost jako by nebyla - neuvažuje se při vytváření polyedru. Inaktivace nerovnosti se obvykle projeví tím, že polyedr se zvětší, někde ho přibude. Rovina, která odsekávala jeho část, se neuvažuje a proto je původně odseknutá část zahrnuta do polyedru. Zkuste postupně

inaktivovat nerovnosti a dívat se, jak se to projeví na polyedru. Zajímavější je možná nerovnosti zpětně aktivovat (zase klepnutím na - nyní tmavý - čtvereček). Někdy se operací odsekne významná část dosavadního polyedru. Odseknutí je nejlépe vidět, pokud hraniční rovina poloprostoru určeného nerovností je vidět "z boku", protože pak má odseknutí výrazný vliv na siluetu polyedru.

Hrana polyedru je úsečka, polopřímka nebo přímka, která je průnikem dvou sousedních stěn. Hrany omezeného polyedru jsou vždy úsečky, neomezený polyedr má alespoň některé hrany tvořeny polopřímkami. Úplná přímka jako hrana se vyskytuje jen zřídka. *Vrchol* polyedru je bod, ve kterém se stýká více hran (obvykle 3 hrany, ale někdy - například u některých pravidelných těles - i více hran). Stejně tak je možno vrchol popsat jako bod, ve kterém se stýkají alespoň 3 stěny (a obvykle právě 3 stěny).

Scéna: *Nerovnosti*

Na rozdíl od roviny je ve třídimenzionálním prostoru nemožné kurzorem vybrat jednoznačně a srozumitelně libovolný bod v prostoru. Pokud ale zamíříte kurzorem na obrázek polyedru, vyberete tím jednoznačně viditelný bod polyedru, tedy jeho bod na přivráceném povrchu. Není možné vybrat bod pod povrchem (ve vnitřku polyedru) nebo na jeho odvrácené straně (pokud jím neotočíte). Ukazuje-li kurzor mimo obrázek polyedru, pak nevybírá žádný bod.

Pokud je knoflík myši stisknut, jsou barvy nerovností stejné jako v předchozí scéně, ale při uvolněném knoflíku jsou voleny jinak. Nerovnosti odpovídající neviditelným stěnám jsou sice zase psány šedě na černém podkladu, ale nerovnosti odpovídající viditelným stěnám jsou psány černým písmem na zeleném podkladu, který však může být buď tmavě nebo světle zelený.

Volba světle nebo tmavě zelené barvy ukazuje, jak jsou nerovnosti odpovídající viditelným stěnám splněny pro bod určený polohou kurzoru. Tmavá barva označuje nerovnost splněnou jako *rovnost*, světlá je splněna jako *ostrá* nerovnost. Pohybujte kurzorem v okně, aniž by byl stisknut knoflík myši. Je-li kurzor mimo obrázek polyedru, není vybrán žádný bod polyedru a proto jsou všechny nerovnosti odpovídající viditelným stěnám světle zelené. Zajeďte nyní kurzorem na polyedr. Nerovnost, odpovídající stěně, ve které je kurzor, nyní bude tmavá.

V zásadě jsou 3 možnosti. Buď je kurzor *uvnitř* některé stěny; pak je tmavě zelená pouze jediná nerovnost, která odpovídá této stěně. Nebo se kurzor dostane na některou *hranu*, ale nikoli na její koncový bod. Jelikož hrana je společná dvěma stěnám, budou nyní tmavě zelené dvě nerovnosti, odpovídající stěnám, které spolu sdílejí uvedenou hranu. Poslední možnost je, že kurzor se dostane na *vrchol* polyedru. Vrchol obvykle patří najednou do právě tří stěn, takže tmavě zelené budou právě tři nerovnosti. V některých situacích ale vrchol patří do více stěn a proto ukazujee-li kurzor na vrchol,

budou někdy tmavě zelené čtyři nebo i více nerovností. Zkuste si například zvolit libovolný vrchol dokonalého osmistěnu nebo dvacetistěnu nebo horní vrchol alespoň 4-bokého jehlanu.

Pokud bychom neměli pouze 3 proměnné a tedy bychom se nepohybovali ve třírozměrném prostoru, jak je tomu v této scéně, ale proměnných by bylo n, pak by vrchol polyedru v n-rozměrném prostoru byl charakterizován tím, že v něm je (alespoň) n nerovností určujících polyedr splněno jako rovnost.

Uvažujme nyní obvyklý případ, kdy každý vrchol leží v právě 3 (resp. n) stěnách. Z příkladů v této scéně je to prakticky vždy u náhodného polyedru, dále u trojbokého jehlanu, krychle a dvanáctistěnu. Pak je vidět, že tento vrchol leží také v právě 3 (resp. n) hranách a hranu vycházející z takového vrcholu určíme tím, že vybereme jednu ze stěn, do kterých vrchol náleží a dovolíme, aby vrcholy hrany *neležely* na vybrané stěně, požadujíce ale při tom, aby ležely ve zbývajících 2 (resp. $n-1$) stěnách určených vrcholem. Na druhém konci takové hrany se nachází vrchol, který leží také ve 2 (resp. $n-1$) stěnách určujících hranu, a navíc v některé další stěně.

Scéna: *Lineární funkcionál v prostoru*

Lineární funkcionál v třídimenzionálním prostoru je zobrazení, které každému bodu $[x, y, z]$ prostoru přiřazuje reálné číslo $Ax + By + Cz$, kde A, B a C jsou pevně daná reálná čísla určující funkcionál.

Neumím zobrazit na obrazovce názorným způsobem obecný lineární funkcionál tak, aby znázorňoval hodnoty funkcionálu ve všech bodech prostoru. Není to ale pro naše účely nutné: podobně, jak tomu bylo v rovině, i v prostoru simplexový algoritmus operuje výhradně na povrchu polyedru, na kterém se hledá maximum funkcionálu. Proto funkcionál budeme znázorňovat tak, že každý bod na povrchu polyedru obarvíme barvou, které v tepelné škále odpovídá hodnotě funkcionálu v uvažovaném bodě. Tepelná škála byla vysvětlena ve scéně o lineárních funkcionálech v rovině.

Ve volbě funkce na ovladači přibyla další funkce **Změň funkcionál.** Jeli zvolena, pohybujte kurzorem po obrazovce se stisknutým knoflíkem myši. Podobně jako jsme jindy rotovali kolem os polyedr, nyní pohybujeme pomyslným vektorem (A, B, C) (není nakreslen) určujícím funkcionál a odpovídajícím způsobem se mění i jeho hodnoty na povrchu polyedru.

Úloha lineární optimalizace se nyní dá popsat tak, že hledáme bod, který je ze všech bodů polyedru nejsvětlejší.

Technický problém v této scéně nastává v okamžiku, kdy je polyedr neomezený, protože na něm nabývají hodnoty funkcionálu libovolně hodnot libovolně blízkých plus a/nebo minus nekonečnu, zatímco tepelná škála používaná pro vizualizaci jeho hodnot je konečná. Problém je vyřešen tak, že nekonečné stěny a hrany jsou oříznuty a z polyedru je zobrazena jen část, která ale obsahuje všechny vrcholy, omezené hrany a stěny a alespoň kus z každé neomezené

stěny a hrany. Je zřejmé, že rostou-li (resp. klesají) hodnoty funkcionálu podél hrany směrem k místu, kde je uříznuta, rostly by dále až do nekonečna (resp. klesaly do minus nekonečna).

Znovu poznamenávám, že neomezený polyedr je kreslen jako dutý, protože veškerá činnost simplexového algoritmu se odehrává na jeho povrchu a vnitřní body by jen činily obrázek méně přehledným.

Scéna: *Pohyb po hranách polyedru*

Tato scéna ukazuje polyedr stejně jako scéna předchozí. Nyní ale zkusíme cestovat po povrchu polyedru tak, že budeme přecházet z jednoho vrcholu do druhého po hranách.

Kotoučkem pohybujeme tak, že změníme volbu **Rotuj polyedr** na **Posuň žeton** a pak klepneme myší na libovolnou hranu, která vychází z vrcholu obsazeného žetonem (pokud není vidět, je třeba rotovat polyedrem, až se objeví). Pohybujte žetonem po polyedru a sledujte barvu pozadí nerovností.

Pokud sedí žeton ve vrcholu, pak tento vrchol splňuje obvykle tři (v n-dimenzionálním případě n) z nerovností určujících polyedr a tři (resp. n) nerovnosti jsou proto tmavě zelené. Vydá-li se po hraně, pak jedna z nerovností začne platit jako ostrá nerovnost (v naší tabulce se stane světle zelenou - sledujte to při animaci). Pohybuje se přitom po přímce, která je průnikem hraničních rovin těch dvou (resp. $n-1$) nerovností, které zůstávají tmavě zelené, zatímco u zbývající nerovnosti, která byla na počátku pohybu tmavě zelená, se rozdíl mezi pravou a levou stranou stále zvětšuje.

Podívejme se nyní na ostatní nerovnosti. U některých se rozdíl mezi levou a pravou stranou při pohybu žetonu stále zvětšuje a nerovnost "platí stále více". Taková nerovnost by zůstala v platnosti, i kdyby se žeton pohyboval až do nekonečna. Pokud ale alespoň v jedné nerovnosti se rozdíl pravé a levé strany v absolutní hodnotě zmenšuje, poklesne po jisté době na nulu - nerovnost začne být splněna jako rovnost. Jakmile nastane první takový okamžik, není již možné dále pokračovat, aniž by bod nevystoupil ven z polyedru. To znamená, že jsme se dostali do jiného vrcholu polyedru. Povšimněte si, že v něm opětovně platí v alespoň 3 (resp. n) nerovnostech rovnost: 2 (resp. $n-1$) jsou ty, které určovaly použitou hranu a třetí (resp. n-tá) je nově splněná rovnost, představující zarážku dalšího postupu.

Ve výjimečných situacích se může stát, že alespoň některý vrchol leží na okraji 4 nebo více stěn (viz osmistěn a dvacetistěn v příkladech). Pokud v takovém vrcholu leží žeton, jsou tmavě zelené více než 3 (resp. n) nerovnosti. Při pohybu po hraně všechny kromě dvou (resp. $n-1$) zesvětlí. Může se také stát, že při pohybu žetonu nastane rovnost současně ve dvou nebo více nerovnostech jiných než ty, které určují hranu. Pak zase budou při zastavení žetonu více než 3 (resp. n) nerovnosti tmavě zelené.

Scéna: *Simplexový algoritmus v prostoru*

Úloha lineárního programování v třídimenzionálním prostoru je nalézt takový bod nebo body polyedru, pro které daný lineární funkcionál nabývá maximální hodnoty. Jak již bylo řečeno výše, znamená to v naší grafické reprezentaci najít nejsvětlejší bod na povrchu polyedru. Tato scéna pracuje s omezenými polyedry, takže řešení vždy existuje.

Vytvořte si nejprve vhodný polyedr a pak, využívajíce různé volby funkce myši, si ho nastavte do vhodné polohy, nastavte podle svých požadavků lineární funkcionál a do zvoleného vrcholu polyedru vložte žeton. Poté stiskněte knoflík **Počítej.** Algovize si provede výpočet, přepne do animačního módu a umožní vám výpočet krokovat nebo zobrazit v běhu a to i s případným vracením. Je také možno vypnout zobrazování funkcionálu a/nebo nerovností.

Simplexový algoritmus pracuje v prostoru v prakticky stejně, jako tomu bylo v rovině:

zvolíme si libovolný počáteční vrchol polyedru (zde zamlčuji, že nalézt pro danou soustavu nerovností nějaký vrchol polyedru jimi definovaného není někdy vůbec lehké, dokonce i sama otázka neprázdnosti takového polyedru může být obtížná; tento problém ale nebudu podrobněji rozebírat);
pokud ve směrech všech hran, vycházejících z vrcholu, funkcionál klesá nebo zůstává konstantní, výpočet končí a aktuální bod je optimální řešení problému, v opačném případě Algovize libovolně zvolí hranu vycházející z bodu, podél níž funkcionál roste, převede žeton po zvolené hraně do jejího druhého konce a tento postup opakuje. Když je nalezeno optimální řešení, žeton zčervená.

Existují různé varianty algoritmu, lišící se ve výběru hrany, podle které žeton postupuje, pokud vhodných hran z vrcholu vychází víc. Volbou hrany je možno v dobrém i špatném smyslu ovlivnit výpočetní rychlost, nikoli ale správnost. Tato problematika zde rozebírána nebude.

Scéna: *Simplexový algoritmus - neomezený polyedr*

Scéna je stejná jako předchozí, ale pracuje výhradně s neomezenými polyedry. Pokud se stane, že narazíme během výpočtu na nekonečnou hranu, podél které hodnota funkcionálu roste, znamená to, že úloha nemá omezené řešení.

Scéna: *Simplexuj si sám v prostoru*

Scéna ukazuje totéž jako předchozí dvě scény - výpočet simplexového algoritmu. Kotoučkem ale nepohybuje na pokyn uživatele Algovize, ale uživatel sám a to stejným způsobem jako ve scéně “Pohyb po hranách polyedru”. Algovize jen kontroluje postup výpočtu a při chybném kroku protestuje jako tomu bylo při uživatelském simplexování v rovině.

Obsah