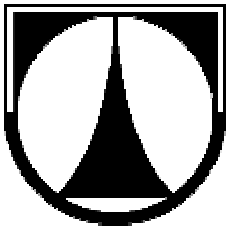
TECHNICKÁ UNIVERZITA v LIBERCI

Fakulta mechatroniky a mezioborových inženýrských studií



DIPLOMOVÁ PRÁCE

Algoritmy pro výpočet minimální kostry grafu

Algorithms for computation minimum spanning trees

Miroslav Novák

Liberec, květen 2003

Anotace

Cílem diplomové práce bylo seznámit se s nejnovějšími metodami řešení problému hledání minimální kostry grafu a porovnat je s klasickými přístupy. Byly vytvořeny výukové aplety a aplikace pomocí programovacího jazyka Java. Aplety byly začleněny do internetových stránek zabývajících se touto problematikou. Tyto stránky a aplikace rozšíří používané pomůcky při výuce diskrétní matematiky a operační analýzy. Závěry diplomové práce ukazují, že nejstarší Borůvkův algoritmus se hodí jako základ pro nejnovější metody řešení problému minimální kostry grafu.

Annotation

The aim of Diploma thesis was understand to new methods of solving minimum spanning tree problem and compare them with standard ways of solving. Applets and applications were made by using programming language Java. Applets were implanted into web pages dealing with these problems. This web sites and applications will enrich teaching aids on the field of discrete mathematics and operation analysis. Results of this Diploma thesis show, that the oldest Boruvka´s algorithm is most useable as the base for new progressive methods of solving minimum spanning tree problem.

TECHNICKÁ UNIVERZITA V LIBERCI

Fakulta mechatroniky a mezioborových inženýrských studií

Studijní program : 2612M - Elektrotechnika a informatika
Studijní obor : 2612T - Automatické řízení a inženýrská informatika

Algoritmy pro výpočet minimální kostry grafu

Algorithms for computation minimum spanning trees

Miroslav Novák

Vedoucí diplomové práce: RNDr. Ladislav Mečíř
Technická univerzita Liberec

Rozsah práce:		
	stran	75
	definic	5
	obrázků	11
	tabulek	4
	grafů	7
	příkladů	5
	CD	1

Prohlášení

Byl jsem seznámen s tím, že na mou diplomovou práci se plně vztahuje zákon č.121/2000 o právu autorském, zejména § 60 (školní dílo).

Beru na vědomí, že TUL má právo na uzavření licenční smlouvy o užití mé DP a prohlašuji, že **s o u h l a s í m** s případným užitím mé diplomové práce (prodej, zapůjčení, apod.).

Jsem si vědom toho, že užít své diplomové práce či poskytnout licenci k jejímu využití mohu jen se souhlasem TUL, která má právo ode mne požadovat přiměřený příspěvek na úhradu nákladů, vynaložených univerzitou na vytvoření díla (až do jejich skutečné výše).

Prohlašuji, že jsem diplomovou práci vypracoval samostatně s použitím uvedené literatury a na základě konzultací s vedoucím diplomové práce a konzultantem.

V Liberci dne 23. 5. 2003 …….

Miroslav Novák

Místopřísežné prohlášení

„Místopřísežně prohlašuji, že jsem diplomovou práci vypracoval samostatně s použitím uvedené literatury.“

V Liberci dne 23. 5. 2003 …….

Miroslav Novák

Poděkování

Za pomoc při řešení diplomové práce bych rád poděkoval vedoucímu práce RNDr. Ladislavu Mečířovi, že mi ochotně umožnil získat cenné odborné informace formou konzultací a technické dokumentace.

Obsah

Seznam použitých veličin a značek

t	[s]	čas
n		počet uzlů
m		počet hran
$O(n^2)$		časová náročnost algoritmu
G		graf
V		množina všech uzlů
E		množina všech hran
h		hrana
u,v		koncové uzly hrany
w(h)		ohodnocení (váha) hrany h
w(E)		cena kostry grafu
T		strom
B		plně se větvící strom
f(v)		list plně se větvícího stromu
$R_x(G)$		řez grafem G
atd.		a tak dále
např.		například
resp.		respektive
tj.		to je
tzn.		to znamená
C		programovací jazyk C
C++		programovací jazyk C++
JVM		virtuální stroj Javy (Java Virtual Machine)
AWT		standardní knihovna Javy
WWW		celosvětová síť (World-Wide Web)
HTML		jazyk pro vytváření dokumentů (HyperText Markup Language)

1. Úvod

Téměř před sto lety se v naší zemi narodili dva později velmi významní matematici Vojtěch Jarník a Otakar Borůvka. Na přelomu let 1925-1926, v době, kdy na jižní a západní Moravě probíhala elektrifikace, se Otakar Borůvka setkal s pracovníkem Západomoravských elektráren Jindřichem Saxelem. Ten jej požádal o pomoc při řešení problému, kudy vést elektrické vedení spojující několik desítek obcí, aby bylo co nejkratší, a tím co nejúspornější. Otakaru Borůvkovi se podařilo daný problém nejen vyřešit, ale také správně formulovat. Na jeho práci navázal Vojtěch Jarník a navrhl jiný a jednodušší postup pro vytvoření požadované konstrukce. Třetí řešení problému, odlišné od předchozích dvou, podal roku 1956 Joseph B.Kruskal. Dlouhou řadu let se v zámoří o Borůvkovi ani Jarníkovi nevědělo, a proto byly tyto algoritmy znovu objevovány.

Přestože jsou známy algoritmy hledání minimální kostry grafu, tento problém stále zůstává v centru pozornosti mnoha odborníků. Jejich snahou je nalézt co nejrychlejší a nejdůmyslnější algoritmus řešící problém minimální kostry nejen obyčejného grafu, ale i různých speciálních grafů.

Cílem diplomové práce je seznámit se s nejnovějšími metodami řešení problému minimální kostry grafu, zejména s článkem popisujícím nalezení minimální kostry grafu v čase, který je skoro jistě lineární. Dále prostudovat klasické přístupy řešení problému minimální kostry grafu a vytvořit odpovídající programové vybavení. Úkolem je vytvořit interaktivní aplety realizující klasické algoritmy hledaní minimální kostry grafu a aplikace pro výpočet minimální kostry grafu pracující dle svých časových náročností. V závěru práce je třeba porovnat jednotlivé algoritmy a zhodnotit dosažené výsledky, které by měly být využívány při výuce diskrétní matematiky a operační analýzy na Technické univerzitě v Liberci.

2. Základní pojmy z teorie grafů

Pro definování a seznámení se s některými pojmy v následujících kapitolách bylo použito studijních materiálů, které jsou v literatuře [6][7].

2.1. Grafy

V následujících kapitolách se budeme setkávat s objektem nazývajícím se graf. Každý graf se skládá z konečné množiny bodů, jež budeme nazývat uzly. Spojnice těchto uzlů budeme nazývat hrany. V grafové terminologii se můžeme setkat s orientovanými a neorientovanými hranami a smyčkami. Každá neorientovaná hrana je plně popsána neuspořádanou dvojicí koncových uzlů $\{u, v\}$. Orientovaná hrana je plně popsána uspořádanou dvojicí uzlů $\{u, v\}$ svých koncových uzlů. V některých aplikacích se můžeme setkat se speciální hranou, která začíná i končí ve stejném uzlu. Takovou hranu nazýváme smyčkou. Jde o situaci, kdy daný prvek systému ovlivňuje sám sebe. Představme si jednoduchý graf, kde uzly představují města a hrany silnice spojující tyto města. Smyčku si nyní můžeme představit jako silnice uvnitř města. Každá smyčka může být dle typu úlohy neorientovaná (popíšeme ji uzlem u k němuž je připojena), nebo orientovaná (popíšeme ji uspořádanou dvojicí (u, u)).

Definice 2.1.1: Definice grafu

Buď V konečná množina. Označme :

$\binom{V}{2} = \{\{u,v\};\ u,v \in V\}$ množinu všech dvouprvkových částí množiny V, V^2 kartézský součin $V \times V = \{(u,v);\ u,v \in V\}$, tj. množinu všech uspořádaných dvojic prvků z V.

G R A F je uspořádaná dvojice $G = (V, E)$ kde $E \subset \binom{V}{2} \cup V^2 \cup V$.

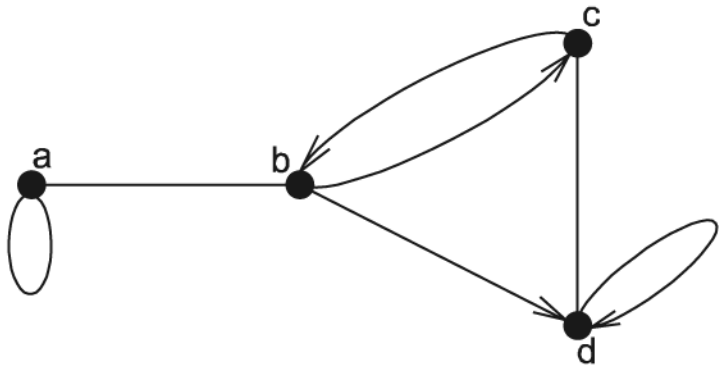
Množina $E \cap \binom{V}{2}$ se nazývá množina neorientovaných hran grafu G.

Množina $E \cap V^2$ se nazývá množina orientovaných hran grafu G.

Množina $E \cap V$ se nazývá množina neorientovaných smyček grafu G.

Množina $E \cap \{(u,u);\ u \in V\}$ se nazývá množina orientovaných smyček grafu G.

Jsou-li dva uzly $u, v \in V$ nějakého grafu *G* spojeny hranou *h*, říkáme, že *u*, *v* jsou sousední . Říkáme, že hrana *h* tyto uzly obsahuje nebo že s nimi inciduje. Je-li *h* orientovaná hrana (*h=(u,v)*), pak uzel *u* se nazývá počáteční uzel hrany *h* a uzel *v* se nazývá koncový uzel hrany *h*. Pro určení orientace hrany se užívá šipky, která směřuje od počátečního uzlu ke koncovému. Na následujícím grafu si ukážeme množiny hran, uzlů a smyček.



Obrázek 2.1.1: Příklad jednoduchého spojitého orientovaného grafu

množina uzlů $V = \{a, b, c, d\}$

množina hran $E = \{\{a,b\}, \{c,d\}, \{b,c\}, \{c,b\}, \{b,d\}, \{d,d\}, a\}$

množina neorientovaných hran $E \cap \binom{V}{2} = \{\{a,b\}, \{c,d\}\}$

množina orientovaných hran $E \cap V^2 = \{\{b,c\}, \{c,b\}, \{b,d\}, \{d,d\}\}$

množina neorientovaných smyček $E \cap V = \{a\}$

množina orientovaných smyček $E \cap \{(u,u);\ u \in V\} = \{(d,d)\}$

Definice 2.1.2: Definice izolovaného uzlu

Uzel $u \in V(G)$ se nazývá izolovaný uzel grafu *G*, jestliže *u* neinciduje s žádnou hranou $h \in E(G)$.

2.2. Neorientované grafy

Definice 2.2.1: Definice neorientovaného grafu

Neorientovaný graf G je uspořádaná dvojice (V, E), kde $V = \{u_1, u_2, u_3, \ldots, u_m\}$ je konečná množina prvků zvaných uzly, E je množina neuspořádaných dvojic uzlů (u_i, u_j), tedy hran. Pro uzly platí, že $u_i \in V$ pro $i = 1, 2, .., n$ a pro hrany platí, že $(u_i, u_j) \in V$ pro $i,j = 1,2,.., m$.

Prostý graf nemá vícenásobné hrany.

Obyčejný graf nemá smyčky ani vícenásobné hrany.

Ohodnocený graf $f : V \rightarrow R$ uzlově ohodnocený graf

$f : E \rightarrow R$ hranově ohodnocený graf

Stupeň uzlu α_i je počet hran incidentních s daným uzlem.

Pro součet stupňů uzlů platí $\sum \alpha_i = 2 \cdot |E|$, kde |E| je počet hran. Graf s vyznačenými stupni uzlu je ilustrován na grafu se šesti uzly a sedmi hranami (viz. Obrázek 2.4.1).

Sled - posloupnost navazujících uzlů

Tah – sled, ve kterém se neopakuje žádna hrana

Cesta – sled, ve kterém se neopakuje žádný uzel

Kružnice - cesta, která začíná a končí ve stejném uzlu

Souvislý graf – každé dva vrcholy lze spojit cestou

Strom – souvislý graf, který neobsahuje žádnou kružnici

Rovinný graf – uzly grafu jsou body roviny a hrany lze reprezentovat spojitými orientovanými křivkami, ležícími v rovině, přičemž mají společné body pouze v koncových bodech. Koncové body křivek jsou uzly grafu. Graf, který není rovinný, nazýváme *prostorový graf* .

2.3. Ohodnocené grafy

V předchozí kapitole jsme se letmo zmínili o ohodnoceném grafu. Podívejme se nyní na podrobnější popis ohodnocených grafů a jejich nejčastější použití. Nejčastěji se používají

hranově ohodnocené grafy, kde každé hraně je přiděleno jistou funkcí ohodnocení. Ukažme si tři praktické použití ohodnocených grafů na následujících situacích:

a) Představme si dopravní síť (například železniční). Tuto síť popíšeme grafem *G*, v němž každému nádraží odpovídá uzel a každé trati odpovídá hrana grafu. Chceme-li pomocí našeho grafu řešit praktické otázky železniční dopravy, je přirozené každé hraně grafu *G* přiřadit reálné číslo, mající význam např. délky tratě.

b) Jestliže je graf *G* grafem elektrického obvodu, tj. uzly odpovídají uzlům obvodu a hrany odpovídají větvím obvodu, pak každé hraně grafu *G* přiřadíme hodnotu příslušného prvku obvodu.

c) Mějme dán libovolný systém, který je třeba převést posloupností operací z výchozího stavu do cílového stavu. Uzly grafu *G* odpovídají jednotlivým možným stavům systému. Hrany popisují operace, jež systém převádějí ze stavu do jiného stavu. V tomto případě můžeme každé hraně přidělit např. dobu trvání příslušné operace, cenu operace apod.

Uvedené příklady nás dovedou k následující definici.

Definice 2.3.1: Definice ohodnoceného grafu

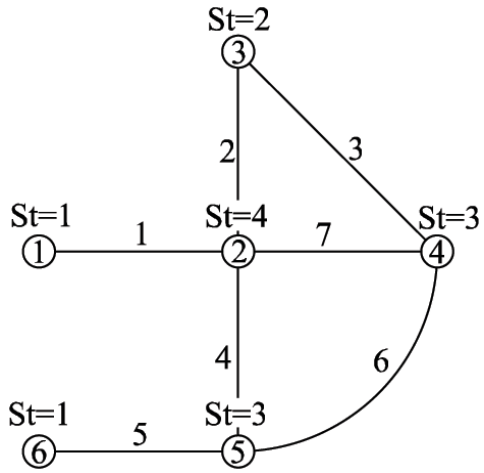
Buď *G* graf. Funkce $w: E(G) \to (0,\infty)$ se nazývá hranovým ohodnocením grafu *G*. Graf se zadaným ohodnocením se nazývá ohodnocený graf.

Pokud hovoříme o ohodnoceném grafu, každá jeho hrana má přidělené jisté ohodnocení. V následujících kapitolách se můžeme setkat s tzv. váhou hrany. Tento pojem je ekvivalentní s ohodnocením hrany. Příklad jednoduchého ohodnoceného grafu je na následujícím obrázku (viz.Obrázek 2.4.1).

2.4. Způsoby zadávání neorientovaných grafů

Níže uvedené způsoby jsou si velmi podobné a výběr správného popisu závisí na povaze úlohy, kterou máme na grafu řešit.

V následujících popisech grafů budeme předpokládat očíslování uzlů i hran. Příklady budou ukázány na jednoduchém neorientovaném ohodnoceném grafu *G* (Obrázek 2.4.1). Graf se skládá z šesti uzlů a sedmi hran. Zkratka *St=4* znamená, že daný uzel je stupně čtyři. V grafu tedy existují čtyři hrany incidentní s daným uzlem.



Obrázek 2.4.1: Neorientovaný ohodnocený graf
s vyznačenými stupni uzlů

2.4.1. Obrázkem

Všechny grafy se dají jednoduše znázornit pomocí obrázku. Grafické znázornění v případě malého počtu uzlů a hran má řadu výhod, kterých žádným jiným zadáním nedosáhneme. Obrázek lze nakreslit velmi rychle. Když jej nakreslíme vhodně, je velice přehledný, názorný a často v sobě ukrývá nějakou vlastnost grafu. Takovýmto obrázkem však graf můžeme zadat pouze člověku a to jen do jeho určité velikosti. Pokud potřebujeme graf zadat počítači, musíme zvolit jiný způsob zadávání.

2.4.2. Výčtem vrcholů a hran

Před středníkem uvádíme počet uzlů a počet hran. Dále uvádíme jmenovitě výčet všech hran.

G:(6,7;{1,2},{2,3},{2,4},{2,5},{3,4},{4,5},{5,6})

Uvážíme-li, že množinové závorky jsou při uvádění hran zbytečné, můžeme graf *G* zapsat pomocí 2 |E(G)| + 2 údajů ve tvaru

G: (6,7; 1,2,2,3,2,4,2,5,3,4,4,5,5,6).

2.4.3. Vyjádřením relace incidence ρ hran a uzlů

a) pomocí množiny set(ρ) všech incidujících dvojic (hrana/uzel). Toto zadání nezachycuje případné izolované uzly. Na prvním místě je uvedená hrana a dále uzel, který je s ní incidentní. Každá hrana je incidentní s dvěma uzly.

set(ρ) = {(1,1), (1,2), (2,2), (2,3), (3,3), (3,4), (4,2), (4,5), (5,5), (5,6), (6,4), (6,5), (7,2), (7,4)}

b) pomocí množiny suc(ρ) všech trojic (hrana, uzel, uzel). Na prvním místě je uvedená hrana a dále dva uzly s ní incidující.

suc(ρ) = {(1,1,2), (2,2,3), (3,3,4), (4,2,5), (5,5,6), (6,4,5), (7,2,4)}

c) pomocí množiny suc(ρ^{-1}) všech uspořádaných x-tic, vyjadřující relaci incidence ρ^{-1}, tj. incidenci uzlů a hran grafu. Na prvním místě je uvedený uzel a dále následuje výčet všech hran incidentních s daným uzlem.

suc(ρ^{-1}) = {(1,1), (2,1,2,4,7), (3,2,3), (4,3,6,7), (5,4,5,6), (6,5)}

d) v případě obyčejného grafu úplnou incidenční maticí. M=[m_{ij}]typu |V(G)|/|E(G)|, kde m_{ij} = 1, je-li $u_i \in h_j$, m_{ij} = 0 všude jinde. Incidenční matice má rozměr n x m, kde n je počet uzlů grafu a m je počet hran grafu.

V našem případě bude mít matice rozměr 6x7. Index řádku si můžeme představit jako číslo uzlu (1..6) stejně jako index sloupce představuje číslo hrany (1..7). Prvek m_{ij} matice M bude roven 1, pokud uzel *i* inciduje s hranou *j*. Ve všech ostatních případech bude m_{ij}=0. Každý sloupec matice *M* reprezentuje jednu hranu a jsou v něm dvě hodnoty 1, které určují uzly incidentní s touto hranou. Součet v *i*-tém řádku matice *M* je roven stupni uzlu u_i.

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

2.4.4. Vyjádřením relace sousednosti ω uzlů grafu

a) pomocí množiny set(ω) všech dvojic (i,j) sousedních uzlů pro $i \leq j$

set(ω) = {(1,2),(2,3),(2,4),(2,5),(3,4),(4,5),(5,6)}

b) pomocí množiny suc(ω) vyjadřující výčet všech okolí všech uzlů grafu

suc(ω) = {(1,2), (2,1,3,4,5), (3,2,4), (4,2,3,5), (5,2,4,6), (6,5)}

c) v případě obyčejného grafu maticí sousednosti S=[s_{ij}] řádu |V(G)|, kde

$$s_{ij} = 1 \ \ pro \ \{i, j\} \in E(G)$$
$$s_{ij} = 0 \ \ pro \ \{i, j\} \notin E(G)$$

V našem případě bude mít matice rozměr n x n, kde n je počet uzlů grafu. Index řádku i sloupce si můžeme představit jako číslo uzlu. Prvek s_{ij} matice *S* bude roven 1, pokud mezi uzly *i* a *j* existuje hrana.

Z matice S můžeme vyčíst některé důležité vlastnosti grafu. Pro neorientované grafy je matice sousednosti symetrická dle hlavní diagonály, zatímco pro orientované grafy není obecně symetrická. Jestliže jsou na hlavní diagonále matice *S* nulové hodnoty, znamená to, že graf *G* neobsahuje žádné smyčky. Součet v *i*-tém řádku matice sousednosti *S* je roven stupni uzlu *i*.

$$S = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

2.4.5. Výčtem uzlů s jejich stupni a sousedy

Na prvním místě je daný uzel, následuje jeho stupeň a výčet uzlů s ním sousedících.

G: (1,1,2,2,4,1,3,4,5,3,2,2,4,4,3,2,3,5,5,3,2,4,6,6,1,5)

3. Problém Minimální kostry grafu

Pro porozumění a definování některých pojmů v následujících kapitolách bylo použito studijních materiálů uvedených v literatuře [4][5].

3.1. Formulace problému

Definice 3.1.1: Definice minimální kostry grafu

Nechť je dán souvislý graf $G = (V, E)$. Pro každou hranu e grafu G je dáno reálné číslo $w(e)$, tzv. ohodnocení hrany e. Mezi všemi kostrami grafu G najděte kostru $H = (V, E')$, pro kterou součet ohodnocení hran $w(H) = \sum w(e)$, kde $e \in E'$, nabývá minimální hodnoty. Kostru H nazveme **minimální kostrou** grafu G a $w(H)$ **cenou kostry** H.

V teorii se většinou ohodnocením hrany grafu rozumí přiřazení reálného čísla. V praxi se však setkáme především s ohodnocením hrany číslem nezáporným. V následujících příkladech budeme pro jednoduchost hodnotit hrany celými kladnými čísly. V praxi se můžeme setkat s příklady hledání minimální kostry grafu, kdy již před výpočtem známe některé hrany, které zcela jistě musí být v minimální kostře grafu nebo naopak. Pak musíme přistoupit k přeznačení ohodnocení hran. Hrany, které se musí zcela jistě nacházet v minimální kostře grafu označíme -1. Hrany, které nesmějí být součástí minimální kostry grafu odebereme a při výpočtu s nimi již nepracujeme. Někdy se můžeme také setkat s úlohou, kdy bychom některé hrany rádi upřednostnili. Musíme tedy analyzovat danou úlohu a přiměřeně snížit ohodnocení těchto hran.

Počet minimálních koster v grafu je dán ohodnocením hran. Mějme souvislý graf, kde každá hrana má jiné ohodnocení $w(e) \neq w(e')$. Různé ohodnocení hran nám zajistí existenci maximálně jedné minimální kostry grafu. V opačném případě si představme graf, který má všechny své hrany ohodnoceny nějakou konstantou K. Každá nalezená kostra takového grafu je jeho minimální kostrou. Ohodnocení každé minimální kostry takového grafu bude $(V-1) \cdot K$.

3.2. Obecný algoritmus nalezení minimální kostry grafu

Obecný algoritmus hledání minimální kostry grafu obsahuje všechny doposud známé přístupy hledání minimální kostry grafu. Algoritmus si ukážeme na souvislém grafu *G=(V,E)*.

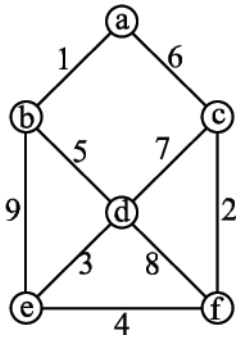
Tento graf má *n* uzlů a *m* hran. Pro každou hranu *e* grafu *G* je dáno celé kladné číslo *w(e)*. Při popisu tohoto algoritmu použijeme proces barvení hran, který nám rozdělí množinu hran daného grafu na hrany, které zcela jistě nemohou být součástí minimální kostry grafu (tzv. červené hrany) a hrany, které zcela určitě tvoří minimální kostru grafu (tzv. modré hrany). Obecný algoritmus používá základních dvou pravidel, pravidla řezu a pravidla kružnice.

Pravidlo řezu: Zvolíme libovolný řez $R_x(G)$, který neobsahuje modrou hranu. Řez $R_x(G)$ v grafu $G=(V,E)$ je množina hran grafu *G*, jejichž jeden koncový vrchol patří do množiny *X*, kde $X \subseteq V$, $0 \neq X \neq V$, a druhý do množiny V-X. Mezi neobarvenými hranami řezu $R_x(G)$ vybereme libovolnou tu, která má minimální ohodnocení a obarvíme ji modře.

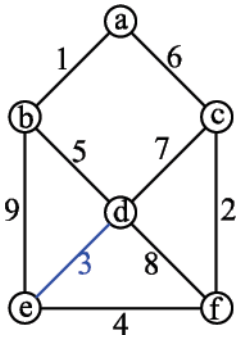
Pravidlo kružnice: Zvolíme libovolnou kružnici, která neobsahuje žádnou červenou hranu. Mezi neobarvenými hranami kružnice vybereme libovolnou tu, která má maximální ohodnocení a obarvíme ji červeně.

Algoritmus opakujeme dokud nejsou obarveny všechny hrany. V každém kroku algoritmu můžeme aplikovat jedno z výše uvedených pravidel a modře obarvené hrany nám vytvářejí modrý les. Obecný algoritmus hledání minimální kostry grafu končí po *m* krocích obarvením všech hran grafu *G*. Po ukončení algoritmu máme dvě množiny hran. Množina modrých hran tvoří minimální kostru grafu. Naopak množina hran obarvených červeně netvoří minimální kostru grafu.

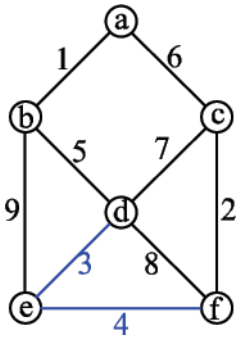
Příklad 3.2.1: Nalezení minimální kostry grafu *G* pomocí Obecného algoritmu s použitím pravidla řezu



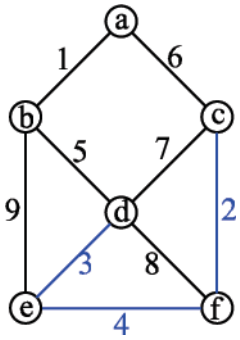
V prvním kroku algoritmu vybereme například řez $R_x(G)$, kde $X=\{e\}$. Řez $R_x(G)$ obsahuje hrany $\{e,b\},\{e,d\},\{e,f\}$. Z těchto hran vybereme hranu s nejmenším ohodnocením a obarvíme ji modře. V našem případě to bude hrana $\{e,d\}$.



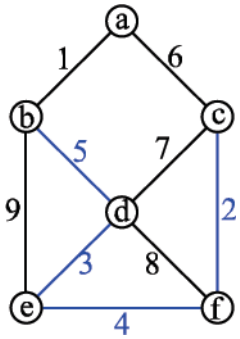
V druhém kroku algoritmu vybereme například řez $R_x(G)$, kde $X=\{e,d\}$. Řez $R_x(G)$ obsahuje hrany $\{e,b\},\{e,f\},\{d,b\},\{d,c\},\{d,f\}$. Z těchto hran vybereme hranu s nejmenším ohodnocením a obarvíme ji modře. V našem případě to bude hrana $\{e,f\}$.



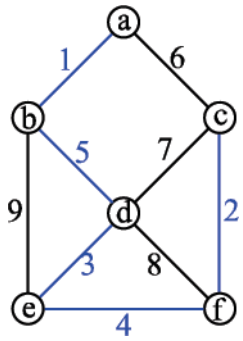
Ve třetím kroku algoritmu vybereme například řez $R_x(G)$, kde $X=\{d,e,f\}$. Řez $R_x(G)$ obsahuje hrany $\{e,b\},\{d,b\},\{d,c\},\{f,c\}$. Z těchto hran vybereme hranu s nejmenším ohodnocením a obarvíme ji modře. V našem případě to bude hrana $\{f,c\}$.



Ve čtvrtém kroku algoritmu vybereme například řez Rx(G), kde $X=\{d,e,f,c\}$. Řez $R_x(G)$ obsahuje hrany $\{e,b\},\{d,b\},\{c,a\}$. Z těchto hran vybereme hranu s nejmenším ohodnocením a obarvíme ji modře. V našem případě to bude hrana $\{d,b\}$.



V pátém kroku algoritmu existuje již jen jediný řez, který neobsahuje modrou hranu. Je to řez $R_x(G)$, kde $X=\{b,c,d,e,f\}$. Řez $R_x(G)$ obsahuje hrany $\{b,a\},\{c,a\}$. Z těchto hran vybereme opět hranu s nejmenším ohodnocením a obarvíme ji modře. V našem případě to bude hrana $\{b,a\}$.

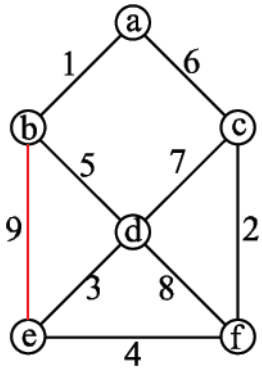


Použitím pravidla řezu získáme po pěti krocích minimální kostru grafu. Cena minimální kostry tohoto grafu je rovna součtu ohodnocení všech hran minimální kostry grafu. V našem případě to bude $w(H) = \sum w(e) = w\{a,b\} + w\{b,d\} + w\{d,e\} + w\{e,f\} +$

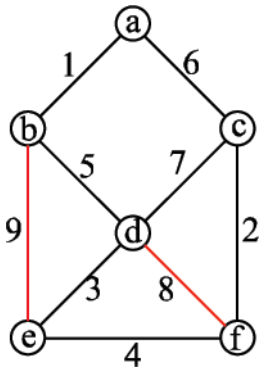
$$+w\{f,c\} = 1+5+3+4+2 = 15$$

V předchozím příkladu jsme používali pravidla řezu. Ve kterémkoliv kroku bychom však mohli použít pravidla kružnice. Vybrali bychom libovolnou kružnici, která neobsahuje červenou hranu a mezi neobarvenými hranami v této kružnici bychom vybrali hranu s největším ohodnocením a obarvili ji červeně. Volba kružnic i řezů je libovolná a jakoukoliv volbou dojdeme ke stejnému výsledku. Minimální kostra grafu *G* je jediná, protože v grafu neexistují žádné dvě hrany se stejným ohodnocením. Ukažme si, že i použitím pravidla kružnice dojdeme ke stejnému výsledku a to dokonce v menším počtu kroků.

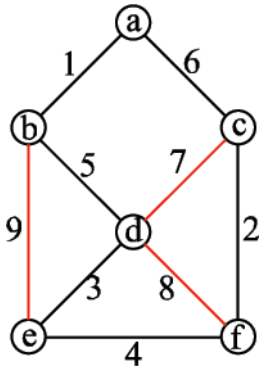
Příklad 3.2.2: Nalezení minimální kostry grafu *G* pomocí Obecného algoritmu s použitím pravidla kružnice



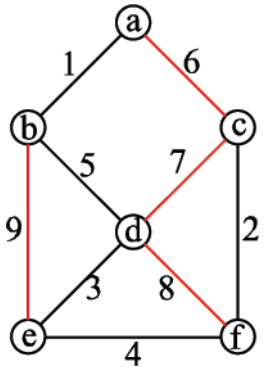
V prvním kroku algoritmu jsou všechny hrany grafu neobarveny. Můžeme si tedy zvolit libovolnou kružnici v grafu. Zvolme například kružnici $K = \{b,d,e\}$. Všechny hrany této kružnice jsou neobarveny. Zvolíme hranu s největším ohodnocením a obarvíme ji červeně. V našem případě to bude hrana $\{b,e\}$, jejíž ohodnocení $w\{b,e\} = 9$.



V druhém kroku algoritmu volíme libovolnou kružnici v grafu, která neobsahuje žádnou červeně obarvenou hranu. Zvolme kružnici $K = \{d,e,f\}$. Tato kružnice opět neobsahuje žádnou obarvenou hranu a tak můžeme červeně označit hranu s největším ohodnocením $\{d,f\}$, jejíž ohodnocení $w\{d,f\} = 8$.



Ve třetím kroku zvolme kružnici $K=\{a,b,d,c\}$. Nejtěžší neobarvená hrana je $\{d,c\}$ s ohodnocením $w\{d,c\}=7$. Obarvíme ji červeně.



V posledním kroku nám již zbývá poslední kružnice neobsahující žádnou červenou hranu. Je to kružnice $K=\{a,b,d,e,f,c\}$. Její nejtěžší hrana je $\{a,c\}$ s váhou $w\{a,c\}=6$. Tuto hranu také obarvíme červeně a jsme hotovi. V daném grafu již neexistuje žádna kružnice, která by neobsahovala červeně obarvenou hranu. Všechny červeně neobarvené hrany tvoří minimální kostru grafu.

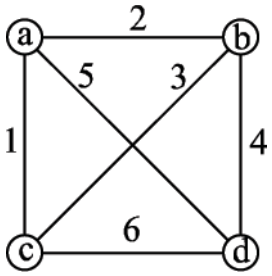
Obecný algoritmus hledání minimální kostry grafu v každém kroku barví hrany. Při použití pravidla kružnice barví hrany červeně, při použití pravidla řezu modře. Algoritmus končí při obarvení všech hran. V předchozích dvou příkladech jsme však ukončili algoritmus již dříve. Obecně můžeme říci, že pokud na daný graf již nemůžeme aplikovat pravidlo řezu, modré hrany takového grafu tvoří minimální kostru. Podobně, neexistuje-li v grafu již žádná další kružnice, která by neobsahovala červenou hranu, zbývající červeně neobarvené hrany tvoří minimální kostru grafu.

3.3. Kruskalův algoritmus nalezení minimální kostry grafu

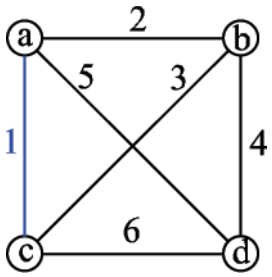
Kruskalův algoritmus bývá velmi často používán pro názorné řešení příkladů hledání minimální kostry grafu. Je velice přehledný a systematický. Ukážeme si jej na souvislém grafu *G=(V,E)* s *n* vrcholy a *m* hranami. Pro každou hranu *e* grafu *G* je dáno celé kladné číslo *w(e)*. Při popisu tohoto algoritmu použijeme opět procesu barvení hran. Na počátku je každá hrana grafu *G* neobarvená. Všechny hrany grafu seřadíme dle jejich ohodnocení do neklesající posloupnosti. Každý uzel grafu *G* považujeme za modrý strom.

V každém z *m* kroků algoritmu rozhodneme právě o jedné hraně, zda ji obarvíme modře či nikoliv. Hrany se zpracovávají dle svého pořadí v neklesající posloupnosti hran. Hranu obarvíme modře právě tehdy, když netvoří kružnici s modře obarvenými hranami. Hrana může tedy být modře obarvena pouze tehdy, když oba její koncové uzly neleží ve stejném modrém stromu. Algoritmus opakujeme dokud není *n-1* hran obarveno modře. Všechny modře obarvené hrany tvoří minimální kostru grafu *G*.

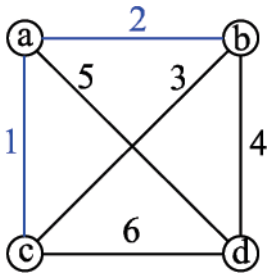
Příklad 3.3.1: Nalezení minimální kostry grafu *G* pomocí Kruskalova algoritmu



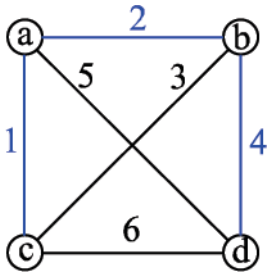
Nejprve seřaďme hrany grafu dle ohodnocení do neklesající posloupnosti $\{a,c\},\{a,b\},\{b,c\},\{b,d\},\{a,d\}$. Každý uzel grafu si představme jako modrý strom. V prvním kroku Kruskalova algoritmu rozhodneme o hraně s nejnižším ohodnocením $\{a,c\}$. Tuto hranu můžeme obarvit modře, neboť netvoří kružnici s modře obarvenými hranami. Její koncové uzly *a*, *c* neleží ve stejném modrém stromu.



V dalším kroku algoritmu budeme rozhodovat o druhé hraně neklesající posloupnosti hran. Hranu $\{a,b\}$ můžeme obarvit modře, protože jejím obarvením nevznikne v grafu modrá kružnice. Oba její koncové uzly neleží ve stejném modrém stromu.



Ve třetím kroku řešíme hranu $\{b,c\}$. Na první pohled vidíme, že jejím obarvením bychom vytvořili v grafu modrou kružnici. Tuto hranu nesmíme obarvit modře. Hrana může být obarvena modře pouze tehdy, neleží-li oba její koncové uzly v témž modrém stromu a to v případě této hrany neplatí.



V dalším kroku rozhodujeme o hraně $\{b,d\}$. Tuto hranu můžeme obarvit modře, neboť netvoří s modře obarvenými hranami kružnici. V tomto kroku bylo již obarveno *n-1* hran modře a můžeme tedy ukončit algoritmus.

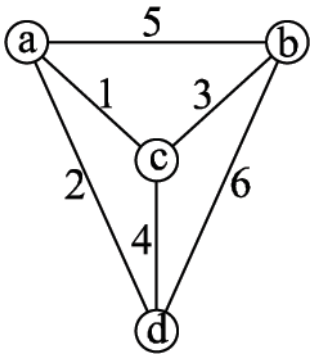
Kruskalův algoritmus v každém kroku rozhoduje právě o jedné hraně. V každém kroku, kdy dojde k obarvení hrany modře, spojuje algoritmus právě dva nejbližší modré stromy. Spojením nejbližších dvou modrých stromů vniká vždy jeden modrý strom. Kruskalův algoritmus lze implementovat s časovou náročností $O(m \cdot \log m) = O(m \cdot \log n)$, neboť $\log m \leq \log n^2 = 2 \cdot \log n$. Algoritmus je výhodné použít pro výpočet minimální kostry z grafu, který obsahuje malý počet hran (je řídký).

3.4. Jarníkův algoritmus nalezení minimální kostry grafu

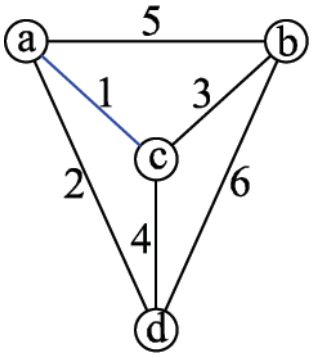
Jarníkův algoritmus si opět ukážeme na souvislém ohodnoceném grafu *G=(V,E)* s *n* vrcholy a *m* hranami. Pro každou hranu *e* grafu *G* mějme dáno ohodnocení hrany celým kladným číslem *w(e)* (může to být i reálné číslo). Při popisu tohoto algoritmu využijeme opět procesu barvení hran. Na počátku algoritmu je každá hrana neobarvená. Zvolme libovolný počáteční uzel *v* z množiny všech vrcholů grafu *V* a obarvěme jej modře. Získáme tím základní počáteční modrý strom $T = (\{v\}, \theta)$. V každém kroku *k, k=1,2, .. ,n-1*, vybíráme ze všech hran incidentních se stromem *T* takovou hranu, která má minimální ohodnocení. Najdeme-li takových hran více, vybereme libovolnou z nich. Tuto hranu obarvíme modře a přidáme ji do stromu *T*. Všechny modře obarvené hrany tvoří minimální kostru grafu *G*.

Jarníkův algoritmus v každém kroku přidává k modrému stromu *T* jednu hranu. Na počátku je modrý strom tvořen jedním uzlem *v*. Tento uzel může být libovolně zvolen z množiny všech uzlů. Zvolením tohoto uzlu určujeme kořen (*root*) stromu *T*. V každém kroku algoritmu přidáváme k modrému stromu hranu s minimálním ohodnocením (list), jejíž jeden koncový uzel leží v modrém stromu *T* a druhý nikoliv. Vybíráme takový řez, jehož množinu *X* tvoří množina uzlů modrého stromu. Tento postup opakujeme dokud strom *T* neobsahuje *n-1* hran.

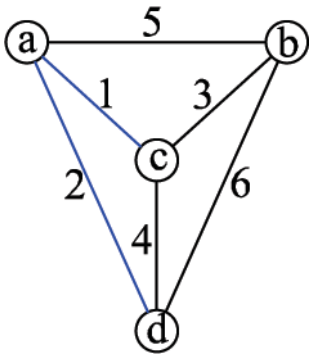
Příklad 3.4.1: Nalezení minimální kostry grafu *G* pomocí Jarníkova algoritmu



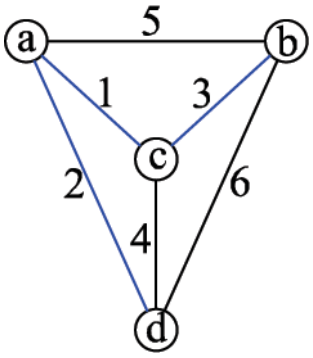
Nejprve zvolíme počáteční uzel a zařadíme jej do stromu *T*. V našem případě zvolme např. uzel *c*. S uzlem *c* jsou incidentní hrany $\{c,a\},\{c,b\},\{c,d\}$. Z těchto hran vybereme hranu s nejmenším ohodnocením. Hrana má jeden svůj koncový uzel ve stromu *T* a druhý mimo. Obarvíme ji modře a zařadíme do stromu *T*.



Strom *T* nyní obsahuje jednu hranu $\{a,c\}$. Hledáme všechny hrany incidentní se stromem *T*. Jsou to hrany incidentní s uzlem *a* $\{a,d\},\{a,b\}$ a s uzlem *c* $\{c,b\},\{c,d\}$. Z těchto čtyř hran má minimální ohodnocení hrana $\{a,d\}$. Jeden z koncových uzlů této hrany leží ve stromu *T* a druhý nikoliv. Hranu tedy můžeme obarvit modře a přidat do stromu *T*.



Strom *T* již obsahuje dvě hrany $\{a,c\},\{a,d\}$. Opět hledáme všechny hrany incidentní s tímto stromem. Jsou to hrany incidentní s uzlem *a* $\{a,b\}$, s uzlem *c* $\{c,b\},\{c,d\}$ a uzlem *d* $\{d,c\},\{d,b\}$. Z těchto hran opět vybereme hranu s minimálním ohodnocením $\{c,b\}$. Tato hrana opět musí splňovat podmínku jednoho svého koncového uzlu ležícího ve stromě T a druhého mimo. Uzel *c* leží ve stromě *T* naopak od uzlu *b*, který není obsažen v *T*. Tuto hranu můžeme obarvit modře a přidat do stromu *T*.



Strom *T* obsahuje tři hrany. Algoritmus můžeme ukončit a konstatovat, že jsme nalezli minimální kostru grafu, protože strom *T* již obsahuje *n-1* hran. V tomto okamžiku bychom mohli zbývající hrany obarvit červeně, protože se nám již nepodaří nalézt hranu incidentní se stromem *T*, která by měla jeden svůj koncový uzel obsažen v tomto stromu a druhý nikoliv.

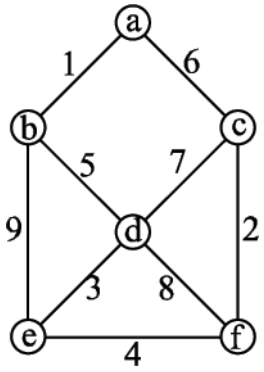
S použitím jednoduchých datových struktur může být Jarníkův algoritmus implementován v čase $O(n^2)$. Algoritmus je výhodné použít při výpočtu minimální kostry grafu pro grafy s velkým počtem hran (řádově n^2) , kde n je počet uzlů grafu. V takovýchto případech pracuje Jarníkův algoritmus rychleji než Kruskalův. Pro malé počty hran je výhodnější použít Kruskalova algoritmu.

3.5. Borůvkův algoritmus nalezení minimální kostry grafu

Nejstarší algoritmus hledání minimální kostry grafu byl popsán Otakarem Borůvkou v roce 1926. Borůvkův algoritmus si ukážeme na souvislém graf G s n vrcholy a m hranami. Pro každou hranu e grafu G nechť je dáno kladné celé číslo $w(e)$ (může být i reálné) takové, že pro každé dvě hrany e, e' grafu G, $e \neq e'$, platí $w(e) \neq w(e')$. Při popisu tohoto algoritmu použijeme procesu barvení hran. Na počátku mějme všechny hrany grafu G neobarvené a každý uzel grafu si představme jako jednoduchý modrý strom. V každém kroku algoritmu zvolíme pro každý modrý strom, mezi všemi hranami incidentními s tímto stromem, hranu s nejmenším ohodnocením. Všechny zvolené hrany obarvíme modře a vždy sjednotíme modré stromy obsahující koncové uzly modře obarvované hrany v jeden modrý strom. Všechny modře obarvené hrany jsou hranami minimální kostry grafu G.

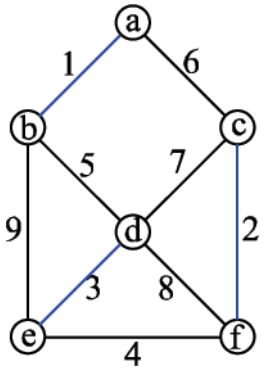
Kruskalův i Jarníkův algoritmus v každém kroku, ve kterém dojde k obarvení hrany modře, spojuje právě dva nejbližší modré stromy v jeden modrý strom. U Kruskalova algoritmu jsou tyto dva stromy určeny vkládanou hranou. Jarníkovým algoritmem je spojován stále se zvětšující jeden modrý strom. Borůvkův algoritmus však v každém kroku spojuje navzájem všechny nejbližší modré stromy.

Příklad 3.5.1: Nalezení minimální kostry grafu G pomocí Borůvkova algoritmu

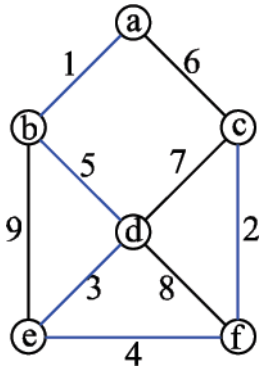


Každý uzel grafu G si představme jako jednoduchý modrý strom. Pro každý modrý strom hledáme všechny hrany s ním incidentní. S uzlem a jsou incidentní hrany $\{a,b\},\{a,c\}$, s uzlem b $\{b,a\}$, $\{b,d\},\{b,e\}$, s uzlem c $\{c,a\},\{c,d\},\{c,f\}$ atd. Z hran incidentních s každým uzlem vybereme hranu s minimálním ohodnocením. Pro uzel a je to hrana $\{a,b\}$, pro uzel b $\{b,a\}$, pro uzel c $\{c,f\}$

atd. Všechny tyto vybrané hrany obarvíme modře a vždy sjednotíme modré stromy obsahující koncové uzly modře obarvované hrany v jeden modrý strom.



V druhém kroku již máme pouze tři modré stromy. Z každého opět najdeme hranu incidentní s minimálním ohodnocením. Ze stromu $\{a,b\}$ to bude hrana $\{b,d\}$, z $\{d,e\}$ to bude hrana $\{e,f\}$ a ze stromu $\{c,f\}$ to bude hrana $\{f,e\}$. Opět tyto vybrané hrany obarvíme modře a sjednotíme modré stromy obsahující koncové uzly barvených hran.



Nyní již máme všechny uzly obsažené v jednom modrém stromu. Nalezli jsme minimální kostru grafu.

Počet kroků Borůvkova algoritmu nemůže být větší než $\log_2 n$, protože v každém kroku se počet stromů v modrém lese sníží alespoň o polovinu. Použitím jednoduchých datových struktur může být Borůvkův algoritmus implementován v čase $O(m.\log n)$.

3.6. Verifikační algoritmus

[8][9][10] V současné době se odborníci snaží nalézt algoritmus, který by pracoval s lineární časovou závislostí O(m). Verifikační algoritmus pracující s touto časovou náročností je znám, a proto se stal základem všech pokrokových algoritmů. Verifikační algoritmus určuje, zda-li daná kostra grafu je minimální kostrou grafu. Tento algoritmus byl v historii studován Robertem E. Tarjanem (1979), Komlósem (1984) a Dixonem, Rauchem a Tarjanem (1992). V současné době se můžeme setkat s modifikací Komlósova algoritmu v článku Valerie Kingové „A Simpler Minimum Spanning Tree Verification Algorithm (July 31, 1995)“.

V této kapitole se jej pokusím přeložit a opravím v něm pár drobných chyb. Dále ho doplním o několik ilustrujících obrázků, které zpřehlední celou problematiku tohoto algoritmu.

3.6.1. Abstrakt

Problém, který je zde diskutován, je problémem, jak zjistit, zda daná kostra grafu je minimální kostrou. V roce 1984 Komlós předložil algoritmus, který požadoval jen lineární počet porovnání, ale nelineární celkový čas potřebný k určení, která porovnání provádět. Zjednodušíme jeho algoritmus a předložíme proceduru pracující v lineárním čase. Procedura používá tabelovaných hodnot několika jednoduchých funkcí, které vypočítáme předem v lineárním čase.

3.6.2. Úvod

Problém určení, zda daná kostra grafu je minimální kostrou, byl studován Tarjanem [T] (1979), Komlósem [Ko] (1984) a nedávno Dixonem, Rauchem a Tarjanem [DRT] (1992). Tarjanův algoritmus z roku 1979 užívá komprese cesty a má skoro lineární čas výpočtu. Komlósův algoritmus byl prvním, který používal lineárního počtu porovnání, ale žádná lineární metoda pro zjištění, která porovnání provádět, nebyla známa. Skutečně, lineární implementace tohoto algoritmu byla považována za nemožnou, viz [Ko] a [DRT]. Jediný lineární algoritmus pro tento problém, [DRT], kombinuje techniky z [T] a [Ko], používaje Komlósův algoritmus pro malé podproblémy pomocí preprocessingu a tabelovaných hodnot.

Tyto verifikační metody a metoda prezentovaná zde využívají faktu, že kostra grafu je minimální kostrou právě, když váha každé nekostrové hrany $\{u, v\}$ je alespoň rovna váze nejtěžší hrany na cestě mezi u a v. Tyto metody hledají nejtěžší hranu v cestě pro každou nekostrovou hranu $\{u, v\}$ v grafu a pak porovnávají její váhu s váhou hrany $\{u, v\}$.

Problém nalezení nejtěžší hrany "kosterní cesty" mezi specifikovanými páry uzlů ("vyšetřované cesty") se objevuje v nedávném nedeterministickém algoritmu hledání minimální kostry grafu prezentovaném Kargerem, Kleinem a Tarjanem. Tento algoritmus je prvním, který nalezne minimální kostru grafu v lineárním očekávaném čase, kde jediné operace povolené na vahách hran jsou binární porovnání. Řešení problému kosterní cesty je nejsložitější částí tohoto nedeterministického algoritmu, který je jinak celkem jednoduchý.

Komlósův algoritmus je zjednodušen použitím následujícího pozorování: Jestliže T je kostrou grafu, pak existuje jednoduchý $O(n)$ algoritmus pro vytvoření plně se větvícího stromu B s nejvýše $2n$ hranami a následujících vlastností:

Nechť *T(x, y)* značí množinu hran v cestě v *T* z uzlu *x* do uzlu *y* a *B(x, y)* značí množinu hran v cestě z listu *x* do listu *y*. Váha nejtěžší hrany v *T(x, y)* je váhou nejtěžší hrany v *B(x, y)*.

Proto stačí použít verzi Komlósova algoritmu pro plně se větvící stromy, která je podstatně jednodušší, než jeho algoritmus pro obecné stromy.

Druhá část tohoto článku dokazuje, že tato část Komlósova algoritmu má implementaci používající tabelovaných hodnot několika jednoduchých funkcí. Tyto tabulky mohou být zkonstruovány v čase lineárním vzhledem k velikosti stromu. Naopak, DRT algoritmus dělí strom na velký podstrom a množství "mikrostromů" o velikosti O(log log n). Na velký podstrom je použita komprese cest. Porovnávací rozhodovací strom potřebný k implementaci Komlósovy strategie pro každou konfiguraci mikrostromu a možnou množinu vyšetřovaných cest v mikrostromu je vypočítán předem a uložen do tabulky. Každý mikrostrom, spolu se svými vyšetřovanými cestami v dané kostře, je zakódován a tabulka je použita pro zjištění, která porovnání je třeba provést.

V další kapitole je popsána konstrukce stromu *B* a je dokázána jeho vlastnost. V kapitole 3 znovu zformulujeme Komlósův algoritmus pro nalezení nejtěžší hrany v každé z *m* vyšetřovaných cest plně se větvícího stromu a popíšeme jeho implementaci.

3.6.3. Vlastnost Borůvkova stromu

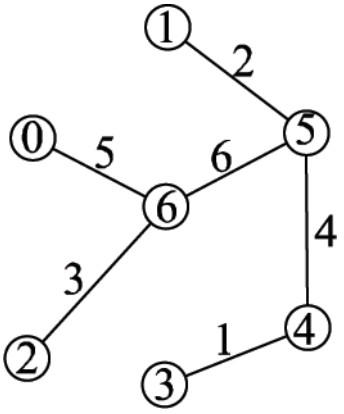
Nechť *T* je kostra grafu s *n* uzly. Strom *B* je stromem komponent, které jsou vytvářeny Borůvkovým algoritmem pro hledání minimální kostry grafu, aplikovaným na *T*.

Borůvkův algoritmus aplikujeme na strom *T=(V,E)*. Na počátku máme *n* modrých stromů skládajících se z uzlů *V* a žádných hran.

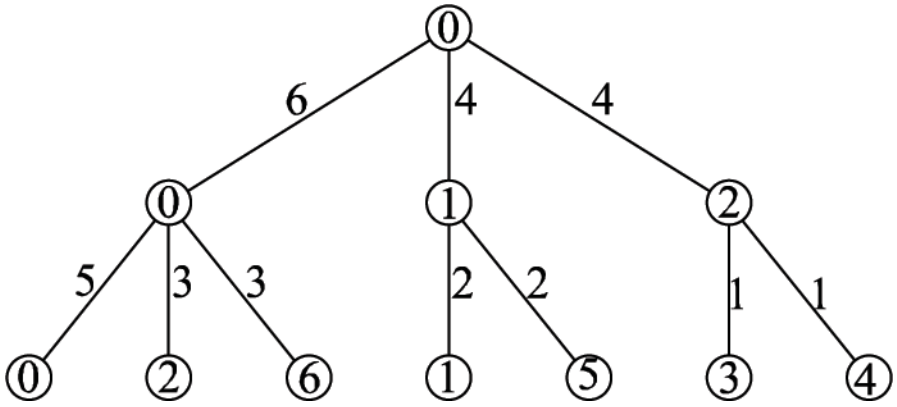
Opakujeme, dokud nemáme jediný modrý strom, tzn., *T*: Pro každý modrý strom určíme hranu s minimálním ohodnocením, která je s ním incidentní. Obarvíme všechny vybrané hrany modře.

Každé opakování těchto instrukcí budeme nazývat fází. Zkonstruujme strom *B*, který je tvořen množinou uzlů *W* a množinou hran *F* tak, že po každé fázi algoritmu přidáme ke stromu *B* obarvené hrany a uzly. Uzly stromu *B* tedy plně korespondují s modrými stromy vytvářenými v průběhu všech fází algoritmu.

Pro každý uzel $v \in V$ z kostry *T* vytvoříme ve stromu B list *f(v)*. Nechť *A* je množina modrých stromů, které jsou spojeny v jeden modrý strom *t* ve fázi *i*. Pak přidáme nový uzel *f(t)* do *W* a do *F* přidáme hranu $\{\{f(a), f(t)\} \mid \forall a \in A\}$. Každá hrana $\{f(a), f(t)\}$ je označena ohodnocením vybrané hrany pomocí *a* ve fázi *i*.



Obrázek 3.6.1: Příklad jednoduché kostry *T* grafu *G*



Obrázek 3.6.2: Plně se větvící strom *B* kostry *T* grafu *G*

Všimněme si, že *B* je plně se větvící strom, tzn., že má jeden kořen (*root*), všechny listy jsou na stejné úrovni a každý vnitřní uzel má minimálně dva následníky.

Protože *T* je strom, *B* může být zkonstruován v čase O(*n*). To by mělo být zřejmé z následujícího: Cena vykonávaní každé fáze je úměrná počtu neobarvených hran stromu v dané fázi. Pokud *T* je stromem, počet neobarvených hran je o 1 menší, než počet modrých stromů. Po každé fázi se počet modrých stromů sníží přinejmenším o polovinu.

Pro každý strom *T*, nechť *T(x, y)* označuje množinu hran v cestě v *T* z uzlu *x* do uzlu *y*.

Dokažme následující větu:

Věta : Nechť *T* je nějaká kostra grafu a nechť *B* je strom zkonstruovaný, jak bylo popsáno výše. Pro každý pár uzlů *x* a *y* v *T*, ohodnocení nejtěžší hrany v *T(x, y)* odpovídá ohodnocení nejtěžší hrany v *B(f(x), f(y))*.

Důkaz : Označme *w(e)* ohodnocením hrany *e*. Nejprve si ukážeme, že pro každou hranu $e \in B\left(f(x), f(y)\right)$ existuje hrana $e' \in T\left(x, y\right)$ taková, že $w(e') \geq w(e)$.

Nechť $e = \{a, b\}$ a nechť *a* je vzdálenější koncový uzel hrany *e* od kořene. Potom *a=f(t)* pro nějaký modrý strom *t*, který obsahuje *x* nebo *y*, ale ne obojí, a *w(e)* je váha hrany vybrané prostřednictvím *t*.

Nechť *e´* je hrana v *T(x,y)* s právě jedním koncovým uzlem v *t*. Protože má modrý strom *t* možnost určit hranu *e´*, $w(e') \geq w(e)$, čímž ukončíme první část důkazu.

Zbývá nám dokázat následující:

Tvrzení : Nechť *e* je nejtěžší hrana v *T(x,y)*. Potom v *B(f(x), f(y))* je hrana se stejnou váhou.

Pro jednoduchost předpokládejme, že hrana s největším ohodnocením je pouze jedna. Důkaz může být jednoduše rozšířen i pro obecný případ.

Jestliže hrana *e* je určena pomocí modrého stromu, který obsahuje *x* nebo *y*, potom hrana v *B(f(x), f(y))* je označena váhou *w(e)*. Naopak předpokládejme, že hrana *e* je určena pomocí modrého stromu, který neobsahuje *x* ani *y*. Takový modrý strom obsahuje jeden koncový uzel hrany *e* a tedy jeden uzel na cestě z *x* do *y*. Protože se jedná o přidaný uzel (ani o *x* ani o *y*), tak musí být incidentní minimálně se dvěma hranami na cestě z *x* do *y*. Potom hrana *e* je těžší hranou a není vybrána, což nám dává spor.

3.6.4. Komlósův algoritmus pro plně se větvící strom

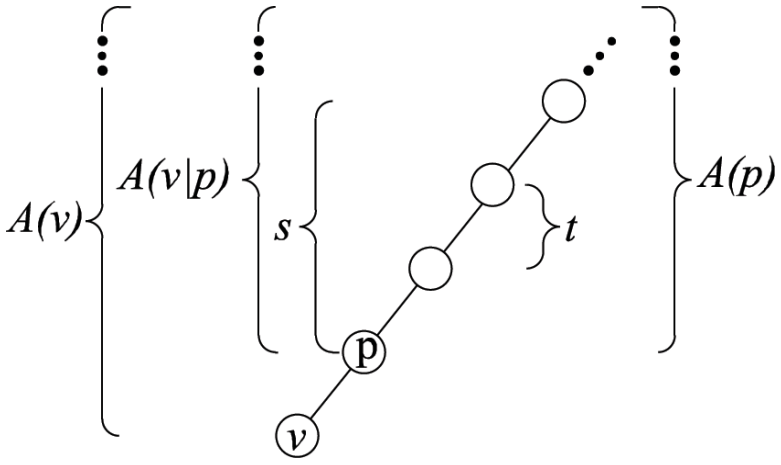
Pro plně se větvící strom s ohodnocenými hranami, *n* uzly a *m* vyšetřovanými cestami mezi páry listů, Komlós předložil jednoduchý algoritmus pro nalezení nejtěžší hrany na cestě mezi každým párem s *O(n log(m+n)/n)* porovnáními.

Dělí každou vyšetřovanou cestu na dvě půlcesty vedoucí z listu až do nejnižšího společného předka páru a hledá nejtěžší hranu v každé půlcestě takto:

Nechť *A(v)* je množinou vyšetřovaných cest, které obsahují *v*, zúžených na interval [*root,v*].

Počínaje v kořenu (*rootu*), sestupujme úroveň za úrovní a pro každý uzel *v* určeme nejtěžší hranu takto:

Nechť *p* je rodičem *v*. Předpokládejme, že známe nejtěžší hranu v každé z cest v *A(p)*. Poznamenejme, že uspořádání vah těchto nejtěžších hran je určeno délkou jim odpovídajících cest, protože pro každé dvě cesty *s* a *t* v *A(p)* platí, že *s* je částí *t*, nebo naopak. Nechť *A(v|p)* je množina restrikcí každé z cest v *A(v)* na interval [*p root*]. Protože *A(v|p)* je částí *A(p)*, uspořádání vah nejtěžších hran v *A(v|p)* je známo. Abychom určili nejtěžší hranu v každé cestě v *A(v)*, potřebujeme jenom porovnat *w({v, p})* s každou z těchto vah. To může být učiněno binárním vyhledáváním. Komlós prokázal, že $\sum_{v \in T} \lg |A(v)| = O(n.\log((m+n)/n))$, což dává horní odhad počtu porovnání potřebných k nalezení nejtěžší hrany v každé polocestě. Nejtěžší hrana každé cesty tak může být nalezena použitím jednoho dodatečného porovnání pro každou vyšetřovanou cestu.



Obrázek 3.6.3: Část plně se větvícího stromu s vyznačenými vztahy *A(p), A(v|p), A(v)*

3.6.5. Implementace Komlósova algoritmu

Implementace Komlósova algoritmu potřebuje použití několika jednoduchých funkcí na slovech délky *O(log n)*, takových, jako posun o specifikovaný počet bitů, bitový OR dvou slov, *log n* zaokrouhlený směrem dolů, součin dvou slov a několik dalších funkcí, které jsou méně běžné a budou popsány níže. Všechny tyto funkce mohou být vypočítány předem v čase *O(n)* a uloženy v tabulce tak, aby se k nim dalo přistupovat v jednotkovém čase. Nejdříve popíšeme datové struktury, které používáme, poté následuje hrubý popis algoritmu a pak jeho implementační detaily.

3.6.5.1. Datové struktury

Nechť *wordsize* je délka slova, kterou předpokládáme [log n] bitů.

Názvy uzlů a označení hran: Následující modifikací Schieberova a Vishkinova [SV] návrhu pojmenujeme uzly [*log n*] bity a hrany *O(log log n)* bity následovně:

Vlastnost názvu: Máme-li dáno nějaké označení hrany *e* a název uzlu na cestě z *e* k nějakému listu, pak hranu *e* můžeme vyhledat v konstantním čase.

Názvy uzlů jsou tvořeny následovně: Listy pojmenujme 0, 1, 2, …, v hloubce jejich výskytu napříč stromem. Každý vnitřní uzel pojmenujeme dle názvu listu s největším počtem 0 v příponě v jeho podstromu.

Pro každou hranu *e*, nechť *v* je její vzdálenější koncový uzel od kořene. Nechť *distance(v)* je vzdálenost uzlu *v* od kořene a *i(v)* je index jedničky umístěné nejvíce vpravo v názvu uzlu *v*. Pak označení hrany *e* je řetězec o délce *tagsize=O(log log n)* bitů. Tento řetězec je dán *<distance(v),i(v)>*.

Ukažme si, proč platí vlastnost názvů, viz. [SV]: Není těžké si povšimnout, že název předka uzlu *w* je dán předponou názvu uzlu *w* pravděpodobně následovanou 1 a pak samými 0. Uzly se stejným názvem jsou spojeny cestou stoupající vzhůru stromem. Proto název uzlu *w* a pozice nejpravější 1 v názvu jeho předka přesně určuje název předka, když jeho vzdálenost od kořene jedinečně určuje předkovu identitu, mezi těmito uzly se stejným názvem. Jakmile jednou nalezneme vzdálenější koncový uzel *v* hrany *e*, potom hrana *e* je jedinečná hrana z uzlu *v* ke svému rodiči.

LCA: Pro každý uzel *v*, *LCA(v)* je vektor délky *wordsize*, jehož *i*-tý bit je 1 tehdy a jen tehdy, když existuje cesta v *A(v)*, jejíž vrchní koncový bod je ve vzdálenosti *i* od kořene. To znamená, že existuje vyšetřovaná cesta s právě jedním koncovým bodem obsaženým v podstromu s kořenem v uzlu *v*. Nejnižší možný společný předek koncových bodů této cesty je ve vzdálenosti *i* od kořene. *LCA* je uloženo v jediném slově.

BigLists a smallLists: Pro každý uzel *v*, nechť je *i*-tá nejdelší cesta v *A(v)* označena $A_i(v)$. Ohodnocení hrany *e* označme *w(e)*. Připomeňme si, že množina cest obsažených v *A(v)* omezená na interval [*kořen,v*] je označována *A(v|a)*. Nazvěme uzel *v big* pokud $|A(v)|>wordsize/tagsize$. V ostatních případech nazvěme uzel *v small*.

Pro každý *big* uzel *v* zavedeme uspořádaný seznam hran, jehož *i*-tým prvkem bude označení nejtěžší hrany v $A_i(v)$, kde $i=1,..,|A(v)|$. Tento seznam nazveme *bigList(v)*. Obdobně můžeme definovat *bigList(v|a)* pro množinu cest *A(v|a)*. *BigList(v)* je uložen v $[|A(v)|/(wordsize/tagsize)]=O(\log \log n)$ slov.

Pro každé *small v*, nechť *a* je jeho nejbližší *big* předek. Pro každý takový uzel *v* zřiďme uspořádaný seznam *smallList(v)*. *I*-tou položkou tohoto seznamu bude označení nejtěžší hrany *e* v $A_i(v)$, nebo pokud hrana *e* leží v intervalu [*a,root*], pak takové *j*, že $A_i(v|a)=A_j(a)$. To znamená, že *j* je ukazatel na záznam *bigListu(a)*, který obsahuje označení hrany *e*. Jakmile se jednou označení hrany objeví ve *smallListu*, všechny další záznamy v seznamu jsou také označení hran. Pro každý *small* uzel *v* si pamatujeme ukazatel na první označení hrany v jeho *smallListu*. *SmallList* je uložen v jediném slově.

3.6.5.2. Algoritmus

Cílem algoritmu je generovat *bigListy(v)* nebo *smallListy(v)* v čase úměrném $\log|A(v)|$ tak, že čas strávený implementací Komlósova algoritmu na každý uzel, nepřekročí v nejhorším případě množství porovnání potřebných na každý uzel. Ukažme si, že pokud uzel *v* je *big*, pak implementační čas je $O(\log \log n)$ a pokud *v* je *small*, je to $O(1)$.

Na počátku máme $A(root)=0$. Zpracováváme strom od kořene směrem k listům, od rodiče *p* ke každému dítěti *v*. Závisle na $|A(v)|$ generujeme buď *bigListy(v|p)* nebo *smallListy(v|p)*. Porovnáme *w({v,p})* s ohodnocením těchto hran použitím binárního vyhledávání na listech a vložíme označení hrany *{v,p}* na příslušné místo *bigListu(v)* nebo *smallListu(v)*. Pokračujeme, dokud nezpracujeme všechny listy.

Nechť *v* je nějaký uzel, *p* jeho rodič a nechť *a* je jeho nejbližší *big* předek. Pro výpočet *A(v|p)* máme následující možnosti:

pokud uzel *v* je *small*:

- Pokud *p* je *small*, vytvoříme *smallList(v|p)* ze *smallListu(p)* v čase *O(1)*.
- Pokud *p* je *big*, vytvoříme *bigList(v|p)* z *LCA(v)* a *LCA(p)* v čase *O(1)*.

pokud uzel *v* je *big*:

- Pokud *v* má *big* předka, vytvoříme *bigList(v|a)* z *bigListu(a)*, *LCA(v)* a *LCA(a)* v čase *O(log log n)*.
  - pokud, $p \neq a$ vytvoříme *bigList(v|p)* z *bigListu(v|a)* a *smallListu(p)* v čase *O(lg lg n)*.
- Pokud *v* nemá *big* předka, pak *bigList(v|p)* $\leftarrow$ *smallList(p)*.

Vložení názvu hrany na její příslušné místo do seznamu:

- Nechť *e={v,p}* a nechť *i* je pozice ohodnocení hrany *w(e)* po porovnání s nejtěžšími hranami z *A(v|p)*. Potom vložíme označení hrany *e* na pozici *i* do *|A(v)|*, do našeho seznamu pro uzel *v* v čase *O(1)* pokud je *v small*, nebo *O(log log n)* pokud je *v big*.

3.6.5.3. Implementační detaily

Výpočet *LCA* je přímý. Nejprve spočteme všechny nejnižší možné předky každého páru koncových uzlů vyšetřovaných cest *m* pomocí algoritmu s časovou náročností *O(n+m)*, viz. [SV] nebo [T2]. Použitím této informace zkonstruujeme vektor *LCA(l)* pro každý list *l* a potom vytvoříme vektor *LCA(v)* pro uzly ve vzdálenosti *i* od kořene, sloučením *LCA* jeho dětí a nastavením *j*-tého bitu do 0 pro všechny $j \geq i$.

Pro realizaci zbývajících operací potřebujeme předběžně zpracovat několik funkcí tak, abychom je mohli vyhledávat v tabulce. Definujme proměnnou *subword*, jejíž velikost je *tagsize* bitů a proměnnou *swnum = [wordsize/tagsize]*. Tzn., že *swnum* je maximální počet *subwords* uložených ve slově. Každý vstup a výstup popsaný níže, je uložen v samostatné proměnné. Symbol • nechť označuje „zřetězeno s“ (př. I • J znamená „I zřetězeno s J“).

$select_r$ bere za vstup $I \bullet J$, kde I a J jsou řetězce o délce r bitů. Výstupem je seznam bitů z J „vybraných“ pomocí I, tzn., nechť $<k_1, k_2, ..>$ je uspořádaný seznam indexů bitů I, jejichž hodnota je 1. Výstupním seznamem je potom $<j_{k1}, j_{k2}, ..>$, kde j_{ki} je hodnota k_i-tého bitu J.

$selectS_r$ bere za vstup $I \bullet J$, kde I je řetězec o maximální délce r bitů a ne více než jedniček v *swnum*, a J je seznam o maximální délce *swnum* podslov. Výstupem je seznam podslov z J, který je „vybrán“ pomocí I. Tzn., nechť $<k_1, k_2, ..>$ je uspořádaný seznam indexů těch bitů z I, jejichž hodnota je 1. Výstupním seznamem je potom $<j_{k1}, j_{k2}, ..>$, kde j_{ki} je k_i-té podslovo z J.

$weight_r$ bere za vstup řetězec o délce r bitů a výstupem je počet bitů nastavených do 1 ve vstupním řetězci.

$index_r$ bere za vstup r bitový vektor s maximálním počtem h jedniček a výstupem je seznam podslov získaných indexy jedničkových bitů ve vstupním vektoru.

$subword_1$ je taková konstanta, že pro $i=1, .., swnum$, $(i*tagsize)$-tý bit je 1 a zbývající bity jsou 0 (tzn., každé podslovo je nastaveno do 1).

Pro všechny tyto funkce je lehce patrné, že předběžné zpracování zabere $O(n)$ času, pokud velikost vstupu není větší než $log\ n+c$, kde c je konstanta. Můžeme vytvořit tabulku pro všechny vstupy délky r tak, že nejprve vytvoříme tabulku pro vstupy o velikosti $r/2$, podíváme se na výsledky dvou polovin a v konstantním čase zpracujeme oba výsledky a vytvoříme záznam.

Např. pro $index_r$, je-li vytvořena tabulka pro $index_{r/2}$, pak můžeme jednoduše zkonstruovat tabulku pro vstupní řetězec délky r v konstantním počtu operací na záznam následovně: Nechť I je první polovina vstupu a J je druhá. Přidej $weight_{r/2}(I)$ ke každému podslovu $indexu_{r/2}(J)$ přidáním $weight_{r/2}(I)*subword_1$. Nechť L je vytvořený řetězec. Potom slučme první $weight_{r/2}(I)$ podslov z $index_{r/2}(I)$ s první $weight_{r/2}(J)$ podslov z L.

Připomeňme si, že *wordsize* má délku $[lg\ n]$ bitů. Nesmíme si dovolit vytvářet tabulku pro $select_{wordsize}$ a $selectS_{wordsize}$, které mají vstupy $2*wordsize$ bitů, protože by tabulka byla

příliš veliká. Ale jak bylo řečeno výše, můžeme vypočítat tyto funkce v konstantním čase, použitím tabelovaného vyhledávání těchto funkcí pro vstupy o velikosti *wordsize/2*, což bylo popsáno výše.

Nyní můžeme provést operace potřebné v naší datové struktuře. (Vynecháme popisy funkcí uvedených níže, neboť mohou být jednoduše vyvozeny z vlastností jejich vstupů.) Ilustrujme si tyto operace na příkladě, kde *wordsize=8* a *tagsize=3*.

1. Určení $|A(v)|$:

   - $|A(v)|=weight(LCA(v))$.

   Příklad: pokud $LCA(v)=(01101110)$, pak |A(v)|=5.

2. Vytvoření *smallList(v|p)* ze *smallList(p)*:

   - $L \leftarrow select((LCA(p),LCA(v))$.
   - $smallList(v|p) \leftarrow selectS(L, smallList(p))$.

   Příklad: nechť $LCA(v)=(01001000)$, $LCA(p)=(11000000)$ a *smallList(p)* je (t_1,t_2). Pak $L=select(11000000,01001000)=(01)$ a $smallList(v|p)=selectS((01),(t_1,t_2))=(t_2)$.

3. Vytvoření *smallList(v|p)* z *LCA(v)* a *LCA(p)*:

   - $smallList(v|p) \leftarrow index(select(LCA(p),LCA(v)))$.

   Příklad: Nechť $LCA(p)=(01101110)$ a $LCA(v)=(01001000)$. Potom $smallList(v|p)=index(select((01101110),(01001000)))=index(10100)=(1,3)$. (Pokud $bigList(p)=(t_1,t_2,t_3,t_4,t_5)$, potom první a druhý záznam *smallListu(v|p)* jsou ukazatele na t_1 a t_3.)

4. Vložení označení *t* na pozici *i* až *j* *smallListu(v|p)* pro vytvoření *smallListu(v)*:

   - Slučme prvních *i*-1 podslov *smallListu(v|p)* s podslovy *i* až *j* z *t*subword1*.

   Příklad: Nechť $smallList(v|p)=(1,3)$ jako v minulém příkladě. Pak *t* je označení *{v,p}*. Abychom mohli vložit *t* na pozici 1 až $j=|A(v)|=2$, musíme spočítat $t*subword1=t*00100100=(t,t)$ následované několika 0 bity, které jsou odstraněny získáním $smallListu(v|p)=(t,t)$.

5. Vytvoření *bigListu(v|a)* z *bigListu(a)*, *LCA(v)* a *LCA(a)*:

   - Nechť $L=select(LCA(a),LCA(v))$.

- Rozložme L na řetězce L_i o délce *swnum* bitů a uložme každé L_i do samostatného slova (poslední řetězec může mít méně bitů).
- Rozložme *bigList(a)* do samostatných slov, kde každé obsahuje *swnum* podslov (poslední může mít méně podslov). Nechť $b_i(a)$ reprezentuje *i*-té slovo *bigListu(a)*.
- Pro každý řetězec L_i, proveďme $selectS(L_i,b_i(a))$.
- Slučme výsledky a vytvořme *bigList(v|a)*.

Příklad: Nechť *LCA(a)=(01101110)*, *LCA(v)=(00100101)* a nechť *bigList(a)* je (t_1,t_2, t_3,t_4,t_5). Potom *L=(01010)*; *L1=(01)*, *L2=(01)* a *L3=(0)*; $b_1=(t_1,t_2)$, $b_2=(t_3,t_4)$ a $b_3=(t_5)$. Následně $(t_2)=selectS((01),(t_1,t_2))$; $(t_4)=selectS((01),(t_3,t_4))$ a $()=selectS(t_5)$. Tedy *bigList(v|a)*$=(t_2,t_4)$.

6. Vytvoření *bigListu(v|p)* z *bigListu(v|a)* a *smallListu(p)*, kde *p* je rodičem *v* a $p \neq a$:
   - Nechť *f* je prvním podslovem *smallListu(p)*, který obsahuje popis hrany spíše než ukazatel. Nahraďme všechny podslova od pozice *f* výše *smallListem(p)*.

   Poznamenejme, že *A(v|p)=A(p)* pouze v případě, kdy $|A(v|p)|>|A(p)|$ a proto *smallList(v|p)=smallList(p)*.

   Příklad: Použitím *a* a *v* z minulého příkladu dostaneme $bigList(v|a)=(t_2,t_4)$. Předpokládejme $smallList(p)=(2,t')$. 2 je ukazatel na druhou položku v *bigListu(a)* a *t´* je označení nějaké hrany ve stromu s kořenem v *a*. Poté $bigList(v|p)=(t_2,t')$.

7. Vložení označení *{v,p}* na příslušné pozice *bigListu(v|p)* pro vytvoření *bigListu(v)*:
   - Řešíme obdobně jako ve výše uvedeném bodě (2), ale musí být vykonán pro každé slovo v seznamu.

3.6.5.4. Analýza

Pokud *v* je *small*, horní cena provedení vložení pomocí binárního vyhledávání je konstantní. Pokud *v* je *big*, $|A(v)|/(wordsize/tagsize) = \Omega(\log n / \log\log n)$, horní cena je *O(log log n)*. Proto implementační cena je *O(log(|A(v)|))*, která je v nejhorším případě úměrná počtu

porovnání potřebných Komlósovým algoritmem k nalezení nejtěžší hrany v *2m* půl-cestách stromu. Sečteno přes všechny uzly nám to dá $O(n.\log(\frac{m+n}{n}))$, což dokázal Komlós.

Při vytváření *LCA* se setkáme pouze s doplňkovými cenami, které zaberou *O(m+n)*, předběžné zpracování tabulek zabere *O(n)* a porovnání nejtěžších hran v každé půlcestě *O(m)*.

K dokončení verifikačního algoritmu nám zbývá porovnat váhu každé nestromové hrany s váhou nejtěžší hrany ve stromové cestě spojující její koncové uzly. Toto porovnání zabere *O(m)*.

3.6.6. Závěr a otevřené otázky

Upravili jsme Komlósův algoritmus pro jednoduchý případ plně se větvícího stromu. Navrhli jsme originální datovou strukturu, která nám dává první algoritmus s lineární časovou náročností.

Stále zůstává otevřenou otázkou, zda-li může být nalezen časově lineární algoritmus pro *pointer machine* (pointer machine - je takový výpočetní model, jehož paměť se skládá z neomezeného počtu registrů nebo záznamů propojených ukazateli. Každý registr může obsahovat libovolné množství přídavných informací. Pro výpočet adresy registru není povolena žádná aritmetika. Jediný přístup k registrům je prostřednictvím ukazatelů). Takový výsledek by přímo vedl k časově lineárnímu algoritmu pro počítače pracující s ukazateli, který by počítal nejbližšího možného předka. Žádné řešení tohoto problému, který se zdá snadnější, není známo.

Je-li dán statický strom, algoritmus hledání nejnižšího společného předka může průběžně zpracovávat zadávané vyšetřované cesty v konstantním čase pro každou z nich. Otevřeným problémem zůstává řešení stromových cest v konstantním čase, kde by dotazované cesty byly zadávány průběžně.

Funkce, které používáme, jsou v určitém smyslu přirozené. Je možné, že mohou být užitečné pro implementaci jiných algoritmů, u nichž zatím není známo, zda mají lineární implementaci, nebo jejichž implementace vyžaduje více specializovaných tabelovaných funkcí, jako například v DRT (viz. DRT, kde jsou uvedeny odkazy na některé z těchto algoritmů).

Na závěr uveďme, že jakékoliv další aplikace věty 1 by mohly být velice zajímavé.

4. Obecný popis Javy

[1][2][3] V průběhu několika posledních let se programovací jazyk Java stal jedním z nejdůležitějších jazyků pro Internet. Jeho vliv na vývoj Internetu však ještě nebyl zcela doceněn. Pomocí tohoto jazyka se Internet změnil na plně interaktivní počítačové prostředí.

Programovací jazyk Java vychází z jazyků C a C++. Základy a koncepci jazyka definovali v roce 1991 pracovníci firmy Sun Microsystems – J. Golsing, P. Naughton. Jejich nově vyvinutý jazyk se jmenoval Oak. Teprve v roce 1995 byl přejmenován na Java. Jazyk je navržen tak, aby aplikace v něm napsané mohly být realizovány na počítačích s různými procesory a pod různými operačními systémy. Má spoustu společných vlastností s jazyky C a C++. Z jazyka C zdědil syntaxi a objektový model vznikl adaptací objektů z C++.

Java může být použita k vytváření dvou druhů programů – *aplikací* a *apletů*.

Aplikace je program, který může běžet samostatně na počítači pod daným operačním systémem.

Aplet je aplikace navržená tak, aby mohla být přenášena pomocí sítě Internet a realizována prohlížečem www stránek, který je kompatibilní s Javou.

Java umožňuje řešit problém bezpečnosti a přenositelnosti dat. Negeneruje totiž spustitelný kód jako překladače ostatních programovacích jazyků, ale generuje tzv. bajtový kód (bytecode). To je vysoce optimalizovaná množina instrukcí navržená tak, aby mohla být spouštěna pomocí run-time systému nazývaného V*irtuální stroj Javy* (Java Virtual Machine). Program v Javě přeložený do bajtového kódu může běžet na různých platformách, protože je nutné implementovat pouze daný virtuální stroj. Implementace virtuálního stroje se pro jednotlivé platformy liší, všechny jsou však schopny interpretovat tentýž bajtový kód. Spuštění každého programu je řízeno virtuálním strojem, čímž je systém chráněn proti nežádoucímu generování jakýchkoli vedlejších efektů mimo systém.

Java je objektově orientovaný jazyk. Základními pojmy pro objektově orientovaný jazyk jsou : *zapouzdření* (encapsulation), *polymorfismus* (polymorphism) a *dědičnost* (inheritance).

Zapouzdření – mechanismus spojující kód a data, se kterými kód manipuluje. Spojením dat a kódu vzniká objekt. Uvnitř objektu lze kód a data deklarovat jako privátní (private) nebo veřejné (public). Privátní kód a data mohou být zpřístupněna pouze kódem uvnitř objektu.

Pokud jsou data a kód veřejná, jsou přístupná i ostatním částem programu. Základní jednotkou zapouzdření je *třída* (class). Třída definuje formu objektu, data i kód.

Polymorfismus – umožňuje zredukovat složitost problému vytvořením společného rozhraní.

Dědičnost – proces, pomocí kterého může objekt získávat vlastnosti jiného objektu → hierarchie. U daného objektu nyní pouze deklarujeme ty vlastnosti, které ho činí jedinečným v dané třídě. Ostatní vlastnosti může zdědit z nadřazené třídy.

4.1. Třídy v Javě

Třídy jsou základem jazyka Java. Používají se k definici objektů. Uvnitř každé třídy se definují kód a data. Kód je obsažen v metodách.

Obecný tvar třídy

```
class jmenotridy {
        //deklarace proměnných instance
        type prom;
        //deklarace metod
        type jmenometody(parametry metody) {
                // telo metody
        }
}
```

příklad deklarace třídy :

```
class Bod {
        int      x;
        int      y;
        int    poradi;
}
```

tato třída má tři parametry typu integer

Metody jsou podprogramy, které umožňují manipulovat s daty definovanými ve třídě. Často také slouží jako rozhraní umožňující přístup k datům.

4.2. Vytvoření objektu

příklad vytvoření objektu:

```
Bod node1;
node1 = new Bod( );
```

Vytvoření objektu se skládá ze dvou kroků. Nejprve se definuje proměnná node1 typu Bod. Tato proměnná je ukazatelem na objekt. V druhém kroku se pomocí klíčového slova new dynamicky alokuje paměť potřebná pro uložení objektu a vrací do proměnné node1 adresu prvního alokovaného bytu paměti.

Celý tento postup lze zkrátit na: `Bod node1 = new Bod( );`

4.3. Konstruktor

Konstruktor inicializuje objekt v okamžiku jeho vytváření. Jazyk Java generuje implicitní konstruktor, který inicializuje proměnné na nulu. Definujeme-li vlastní konstruktor, tak implicitní nebude použit.

4.4. Mechanismus uvolňování místa v paměti

Tento mechanismus se nazývá *garbage collector*. Uvolňuje paměť automaticky bez zásahu uživatele. To lze považovat za výhodu ve srovnání s jazykem C, kde programátor musí uvolňovat paměť programem, pomocí příkazu free.

Garbage collector pracuje tak, že pokud není k dispozici odkaz na objekt v paměti (např. z globálních proměnných), předpokládá se, že objekt neexistuje a z paměti je uvolněn.

4.5. Vytvoření apletu

Obecná struktura apletu:

```
import java.awt.*;
import java.aplet.*;

public class kostra extends Applet {
        public void init( ) {
                // inicializace apletu
        }
        public void start( ) {
                // spuštění nebo obnovení apletu
        }
```

```
        public void stop ( ) {
                // pozastavení apletu
        }
        public void destroy ( ) {
                // operace nutné ke korektnímu ukončení apletu
        }
        public void paint (Graphics g) {
                // obnovení obsahu okna
        }
}
```

Nejprve se pomocí příkazu import importují třídy AWT a Applet. Aplet komunikuje s uživatelem prostřednictvím funkcí ve třídách AWT, které podporují interaktivní grafické prostředí. Každý aplet musí obsahovat třídu Applet. Potom se deklaruje třída kostra, která je potomkem třídy Applet, čili dědí její vlastnosti. Tato třída musí být veřejná (public), aby byla zpřístupněna vnějšímu kódu.

4.6. Inicializace a ukončení apletu

Jako první se volá metoda init( ), pomocí které se provádí inicializace proměnných a počáteční aktivity aplikace.

Poté se zavolá metoda start( ). Tato metoda se také provádí po restartování apletu, pokud byl pozastaven. K tomu může dojít např. tehdy, vrátí-li se uživatel na předcházející www stránku obsahující aplet.

Metoda paint( ) se volá při požadavku překreslení okna apletu. Metoda má jeden parametr typu Graphics, který obsahuje grafický kontext, jenž popisuje grafické prostředí.

Další metoda stop( ) se volá při opuštění stránky s apletem a pozastaví ho. Tuto metodu lze použít k pozastavení kteréhokoliv podřízeného procesu vytvořeného apletem. Metoda však neprovádí ukončení apletu, může totiž být znovu restartován metodou start( ), při návratu uživatele na stánku obsahující aplet.

Pro provedení všech nezbytných operací k ukončení apletu se používá metoda destroy( ).

4.7. Kreslení v Javě

Pro práci s grafikou v Javě se používá základní třída java.awt.Graphic. Její grafické možnosti jsou velmi omezené, ale pro běžné požadavky jako např. vykreslení přímky, obdél-

níku, kružnice, textu atd. plně postačující. Při náročnějších požadavcích na grafiku lze použít třídu java.awtGraphics2D.

Okamžik vykreslení na obrazovku můžeme určit pomocí metody repaint ( ) nebo update( ) ze třídy Component. Metoda repaint( ) volá metodu update( ), která vymaže komponentu, na kterou kreslíme tím, že přes ni překreslí obdélník v barvě pozadí. Potom nastaví grafický kontext g na barvu popředí a nakonec zavolá metodu paint( ), která provede vykreslení na obrazovku.

Není nutné kreslit pouze v metodě paint( ). Grafický kontext lze totiž získat pomocí metody getGraphics( ), ale není to příliš doporučováno, protože toto řešení s sebou nese určité problémy.

Při grafickém řešení apletů jsem narazil na zajímavý problém, jak na plochu přidat nový grafický objekt, aniž by došlo k vymazání dosavadních vykreslených objektů. V literatuře se doporučuje překrýt metodu update( ) tím, že se v ní volá přímo metoda paint( ).

```
Public void update(Graphics g) {
        paint(g);
}
```

Toto opravdu lze použít, ale nastává problém při manipulaci s oknem, ve kterém je aplet zobrazován (např. minimalizace okna). Dochází k přemazání a zůstane vykreslen pouze poslední nakreslený objekt. Tento problém jsem vyřešil tak, že se všechny nakreslené objekty ukládají do paměti a v metodě paint( ) se cyklicky vykreslují.

5. Programové vybavení

Programové vybavení bylo zhotoveno v programovacím jazyce Java. Tento jazyk vychází z koncepce jazyků C a C++. Z programovacího jazyka C Java zdědila syntaxi a objektový model převzala od jazyku C++. Hlavní výhodou tohoto programovacího jazyku je možnost prezentovat naprogramované aplety a aplikace pod různými operačními systémy a procesory. Za programovací prostředí byl zvolen produkt firmy Borland JBuilder 8 Personal. Toto programovací prostředí je volně ke stažení na webové stránce firmy Borland http://www.borland.com/products/downloads/download_jbuilder.html. K používání této volně distribuované verze je zapotřebí se zaregistrovat na stránkách firmy Borland a následně pak obdržet pomocí e-mailu aktivační klíč, který dovolí instalaci JBuilderu na počítač. Minimální hardwarové požadavky pro správný běh programu jsou : Intel Pentium II 233 MHz nebo kompatibilní, minimálně 256MB operační paměti (doporučeno je 512MB) a operační systém Microsoft® Windows 2000 (SP2), XP, nebo NT 4.0 (SP6). Všechny aplety a aplikace byly naprogramovány na počítači s procesorem Intel Celeron 700MHz s operační pamětí 384MB.

5.1. Aplety

Pomocí JBuilderu byly naprogramovány tři základní aplety ilustrující základní tři algoritmy hledání minimální kostry grafu na spojitém ohodnoceném grafu. Tyto aplety jsou programy, které se používají především na WWW stránkách včleněním do jejich HTML kódu. To znamená, že se nespouštějí přímo jako aplikace, ale spouští se otevřením HTML dokumentu pomocí WWW prohlížeče, který umí s aplety pracovat. U apletů je zachovávána vysoká bezpečnost, a proto v nich můžeme použít jen určitou podmnožinu možností programovacího jazyka Java. Aplety například nemohou číst některé systémové proměnné, navazovat síťové spojení na jiný než domovský server a především zapisovat do souborů v počítači, kde je aplet spuštěn. Tato omezení se mohou v různých verzích JDK a různých nastaveních *security-manageru* zpřísňovat nebo naopak zmírňovat a také je mohou měnit i samotné prohlížeče WWW stránek. Protože je aplet spouštěn prostřednictvím WWW prohlížeče, může používat pouze knihovny, které má k dispozici. Zcela funkční aplet přeložený na našem počítači může být spouštěn prohlížečem, který nemá implementovanou podporu Java Core API a jednoduše nebude fungovat. V našich apletech budeme výhradně používat knihovnu AWT, protože podporu všech běžných metod z AWT mají současné WWW prohlížeče implementovánu.

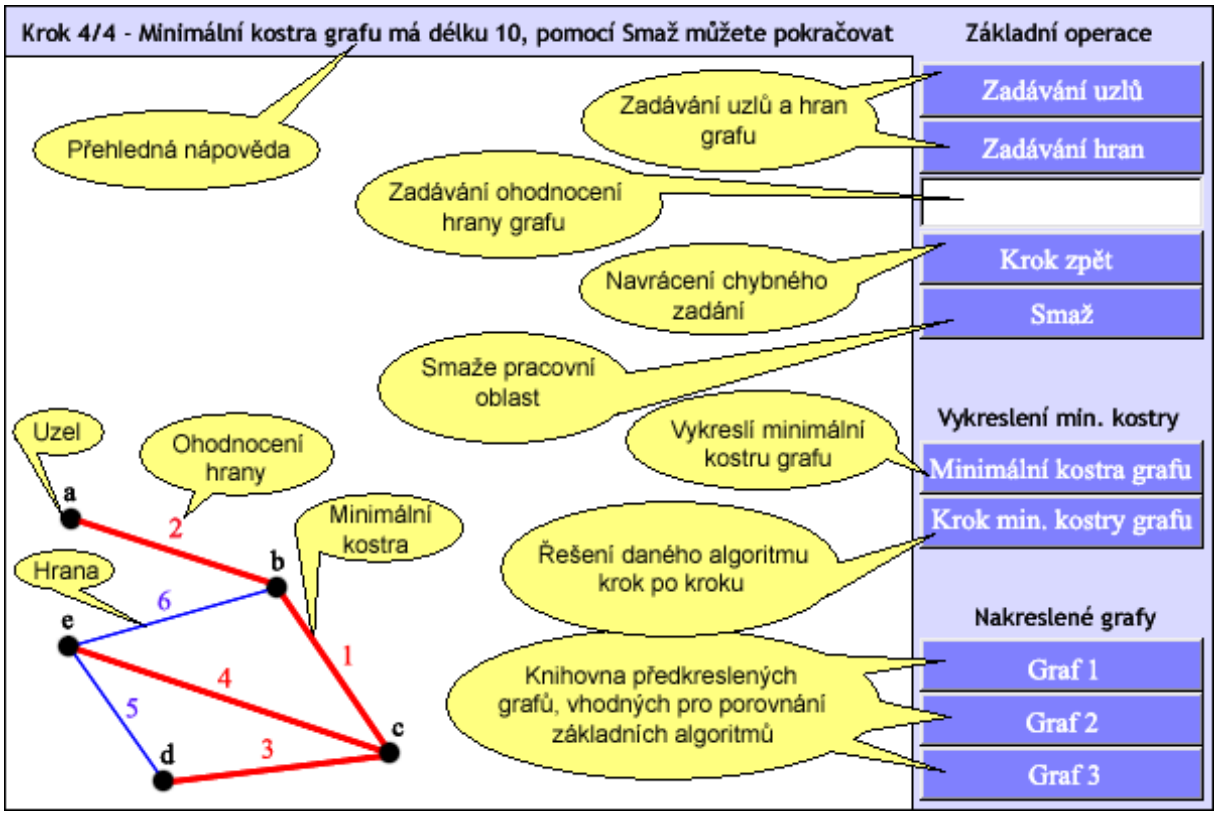
Kdyby WWW prohlížeč spoléhal na podporu Java Core API, kterou by neměl implementovánu, nemusí být vše ztraceno. Existují totiž *pluginy*, které lze do prohlížeče doinstalovat. Myslím si však, že většina uživatelů sítě internet nebude ochotna kvůli apletu instalovat nové *pluginy*, a proto je jednodušší se poněkud omezit a vystačit se standardní knihovnou AWT. Snad posledním problémem apletů jsou sami výrobci WWW prohlížečů. Musíme se smířit s nejednotným chováním apletů, protože každý prohlížeč pracuje s aplety jinak.

5.1.1. Návrh apletů

Hlavním požadavkem při návrhu grafické podoby apletu byla jeho jednoduchost a přehlednost. Uživatel by se neměl zabývat studováním složité obsluhy apletu, ale po velmi krátkém seznámení by jej měl plnohodnotně ovládat. Ke snadnému ovládání velice napomůže přehledná propracovaná nápověda, která uživatele povede při práci. Žádný uživatel není neomylný a proto musí být v apletu možnost provedení oprav chybných zadání. Součtem všech požadavků byl navržen aplet skládající se ze tří hlavních částí. První část je ovládací. Obsahuje základní ovládací prvky vstupu a výstupu. Druhou částí apletu je panel, na kterém je zobrazována přehledná nápověda a ve finální fázi i výsledky výpočtů. Poslední částí je plátno (*Canvas*), které slouží k zadávání grafů a zobrazování výsledků. Aplet je umístěn v HTML kódu a proto bylo třeba důmyslně promyslet jeho velikost. Uživatelé internetu mohou mít nastavená různá rozlišení grafických adaptérů a bylo by nešťastné, kdyby se jim aplet nevešel na obrazovku monitoru a museli by s ním pohybovat (pomocí tzv. *scrollbaru*). Proto byl navržen aplet o šířce 600 obrazových bodů na 400 obrazových bodů.

5.1.2. Ovládání apletů

Ovládání apletů je velice jednoduché a intuitivní. Na následujícím obrázku (Obrázek 5.1.1) je zobrazena pracovní plocha apletu se všemi ovládacími prvky. Tento obrázek byl získán okopírováním obrazovky (pomocí tzv. *print screen*) přímo z počítače, oříznut a doplněn komentáři popisujícími jednotlivé ovládací prvky.



Obrázek 5.1.1: Pracovní plocha apletu se všemi ovládacími prvky

Nejprve si ukažme základní části apletu. Bílou podkladovou barvou je vyznačeno plátno, na které se kreslí graf. Vínovou barvou v pozadí jsou označeny zbylé dvě části apletu. Pod horním okrajem je neustále zobrazována nápověda, která se mění dle uživatelských zásahů. Při pravém okraji apletu nalezneme sloupec ovládacích prvků. Tlačítka „zadávání uzlů“ a „zadávání hran“ slouží k přepínání pracovního režimu. Při pracovním režimu „zadávání uzlů“ může uživatel libovolně rozmísťovat kliknutím levého tlačítka myši na plátně jednotlivé uzly grafu. Tyto uzly jsou automaticky pojmenovány písmeny abecedy vyjma písmen s diakritickými znaménky. Takto může být na plátně rozmístěno až 26 uzlů (počet písmen abecedy). Uzly nejdou pokládat přes sebe ani ve vlastní těsné blízkosti. To samé platí při okrajích plátna. Máme-li již rozmístěny potřebné uzly grafu, můžeme přistoupit k jejich propojení ohodnocenými hranami. Musíme se přepnout do režimu zadávání hran pomocí tlačítka „zadávání hran“. Každá hrana se zadává trojicí informací. První z nich je ohodnocení hrany. Tato informace se píše do bílého pole pod tlačítko „zadávání hran“. Ohodnocením hrany rozumíme celé kladné číslo. V případě, že se pokusíme zadat něco jiného, aplet nás na to upozorní a zadání budeme muset opakovat. Druhou informací potřebnou ke správnému zadání

hrany je počáteční uzel. Kliknutím levým tlačítkem nad uzlem grafu určíme počáteční a pak koncový uzel hrany. Hrana se nám vzápětí vykreslí na plátně i s jejím příslušným ohodnocením. V případě, že jsme hranu zadali špatně a chceme se vrátit o krok zpět, stiskneme tlačítko „Krok zpět“. Kolikrát je toto tlačítko stisknuto, o tolik kroků se vrátíme zpět (obdobně lze vrátit špatné zadání uzlů). Máme-li již zadán kompletní spojitý hranově ohodnocený graf, můžeme přistoupit k výpočtu minimální kostry grafu. Můžeme si vybrat, jak si přejeme kostru počítat. Stisknutím tlačítka „Minimální kostra grafu“ proběhne výpočet a na plátně se v nakresleném grafu červeně vyznačí minimální kostra grafu. Na panelu nápovědy se zobrazí její celková délka a informace o možném pokračování. V případě chybně zadaného grafu (nespojitého) vás po výpočtu aplet upozorní, že minimální kostru zadaného grafu nelze nalézt. Druhý způsob výpočtu je mnohem názornější a slouží především pro pochopení algoritmu. Uživatel opětovným stisknutím tlačítka „Krok min. kostry grafu“ provádí výpočet minimální kostry po jednotlivých krocích algoritmu. V každém kroku se zvýrazní přidávaná hrana ke kostře grafu. V panelu nápovědy je zobrazeno číslo kroku z celkových kroků potřebných pro výpočet minimální kostry grafu pomocí daného algoritmu. Po posledním kroku algoritmu je spočteno celkové ohodnocení minimální kostry (cena kostry) a zobrazeno opět v panelu nápovědy. Nyní nám jen zbývá vysvětlit si poslední tři tlačítka. Pod každým se skrývá nakreslený graf, u kterého si může uživatel spočíst minimální kostru. Samozřejmě se dají předkreslené grafy modifikovat (přidat několik uzlů a hran navíc).

Na závěr bych chtěl upozornit na barevnou odlišnost apletů od popisovaných výpočtů algoritmů. V kapitole 3.2, 3.3, 3.4 a 3.5 byly vysvětleny základní algoritmy hledání minimálních koster grafů pomocí barvení hran. Červeně obarvená hrana znamenala, že daná hrana zcela jistě nepatří do minimální kostry grafu na rozdíl od modře obarvených hran, které tvořily minimální kostru grafu. V apletu je tomu naopak. Všechny hrany jsou kresleny modrou barvou a při výpočtu jsou hrany patřící do minimální kostry obarveny červeně.

5.1.3. Programová realizace apletů

Aplety se skládají ze tří tříd (viz. 4.1). Třída Kostra je základní třídou, která dědí vlastnosti své nadtřídy, třídy Aplet. Tato třída obsahuje hlavní tělo programu. Aplety dále používají dvě definované třídy. Třídy Uzel a Hrana. V těchto třídách se definují proměnné uzlu resp. hrany. Z hlavního programu tak lze přistupovat k jednotlivým bodům resp. hranám jako

k celku, nebo pomocí "tečkového" operátoru k jednotlivým proměnným. Toto členění je velice výhodné, protože je tak jasně dáno, co se týká bodu a co hrany.

Ukázka konstrukce třídy Uzel.

```
class Uzel {
    int      x;
    int      y;
    int    poradi;
    int    komponenta=0;
}
```

Každý uzel je popsán čtyřmi proměnnými. V proměnné pořadí je uloženo číslo uzlu, které získal při umístění na plochu (je to jeho pořadové číslo, ve kterém byl zadán). Proměnná komponenta nese informaci o příslušnosti každého uzlu ke komponentě (modrému stromu) grafu. Ve zbylých dvou proměnných jsou uloženy souřadnice uzlu.

Obdobně je popsána i každá hrana grafu. Proměnné popisující každou hranu nesou informace o jejím počátečním a konečném uzlu, o souřadnici středu hrany a jejím ohodnocení.

Dále se ve všech apletech používají dvě globální celočíselné proměnné, které nesou informaci o aktuálním počtu hran a uzlů. Tyto proměnné se mění dle množství zadaných uzlů a hran. Hrany i uzly jsou uloženy v jednorozměrném poli Hran, resp. Uzlů.

Hlavní třída Kostra se dělí na metody, tedy podprogramy realizující funkce apletu.

V apletech jsou zpracovány následující metody.

Metoda reagující na událost stisku levého tlačítka myši

deklarace metody:

```
void  this_mousePressed(MouseEvent e)
```

V této metodě se provádějí všechny operace vyvolané stiskem tlačítka myši, tedy zadávání uzlů do grafu a výběr počátečního a koncového uzlu hrany (u Jarníkova algoritmu přibývá ještě možnost výběru počátečního uzlu, od kterého algoritmus počítá minimální kostru grafu).

Metoda reagující na událost stisku tlačítka (buttonu)

Deklarace metody:

```
void button1_actionPerformed(ActionEvent e)
```

Třída kostra u Kruskalova i Borůvkova apletu obsahuje devět těchto metod, protože tyto aplety ovládáme devíti funkčními tlačítky. U Jarníkova apletu máme o jedno funkční tlačítko víc (tlačítko pro výběr počátečního uzlu, z kterého se má začít počítat minimální kostra grafu). Funkce jednotlivých tlačítek byla vysvětlena v kapitole 5.1.2.

Metoda kreslení

Deklarace metody:

```
public void paint(Graphics g)
```

Metoda paint( ) slouží ke grafickému vykreslování na obrazovku (viz kapitola 4.7). V apletech byly použity pro kreslení další tři metody. Metoda kresliBod( ) pro vykreslování uzlů grafu, metoda kresliHranu( ) k vykreslení hran grafu a nakonec metoda kresliKostru( ) k vykreslení výsledné kostry grafu.

Všechny tyto metody se volají z metody paint( ). Toto rozdělení na grafické podmetody je důležité hlavně pro udržení přehlednosti programu.

Procedura mazání

Deklarace procedury:

```
public void smaz()
```

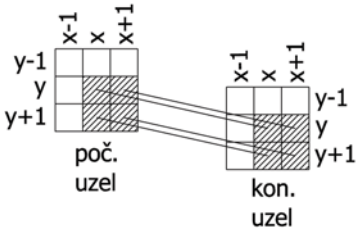
Procedura smaz() slouží k smazaní všech proměnných potřebných pro správný běh programu, před novým zadáním grafu.

Uživatel zadává všechny informace pomocí levého tlačítka myši a numerické klávesnice. Proto je zapotřebí rozdělit aplety do dvou funkčních režimů (režim zadávání uzlů a režim zadávání hran). V apletech jsou zavedeny dvě logické proměnné pro správu těchto režimů. Při spuštění apletu či vymazání všech dat je přednastaven režim zadávání uzlů (pokud nemáme zadány minimálně dva uzly, je nesmyslné zadávat hrany). Po zadání minimálního počtu uzlů potřebných pro vytvoření hrany (počáteční a koncový uzel) se můžeme přepnout do režimu

zadávání hran a to hned dvěma způsoby. Zadáním ohodnocení hrany do textového pole anebo stisknutím tlačítka zadávání hran. Tyto dva režimy práce apletu jsou velice důležité pro mazání případných chybných zadání. Je-li aplet v režimu zadávání uzlů, opakovaným stisknutím tlačítka *smaž* se odebírají sestupně uzly dle pořadí jejich zadání. Obdobně pracuje i odebírání hran. Uzly i hrany jsou tedy zadávány do fronty LIFO (last in first out), u které poslední přidaný prvek se odebírá jako první.

Pro vykreslování zadaných uzlů a hran v předem zmíněných režimech práce, byla použita základní třída pro práci s primitivní grafikou, třída Graphics. Tato třída obsahuje pět základních grafických primitiv – úsečka, obdélník, ovál, kruhová výseč, polygon (uzavřená lomená čára) a polyline (lomená čára). V apletech bylo použito úsečky, oválu a obdélníku. Po zadání uzlu levým tlačítkem myši zjistíme souřadnici, na kterou chceme umístit uzel (x=e.getX(); y=e.getY()). Nastavíme barvu uzlu pomocí příkazu g.setColor(Color.black) a uzel umístíme na předem získané souřadnice pomocí příkazu g.fillOval(int x, int y, int šířka, int výška). Tento ovál je zadán pomocí obdélníku, který jej ohraničuje. Zadáváme tedy souřadnice levého horního rohu (x, y) a jeho šířku a výšku. V našem případě g.fillOval(x-5,y-5,10,10). Nyní ještě potřebujeme zadaný uzel popsat. To provedeme příkazem g.drawChars(char [] název, int offset, int délka, int x, int y). Souřadnice x a y označují počáteční bod, od kterého se bude název vypisovat. Výpis názvu lze omezit proměnnou délka.

Obdobně pracujeme i s vykreslováním hran. Opět získáme počáteční a koncovou souřadnici hrany stisknutím levého tlačítka myši nad počátečním a koncovým uzlem. Úsečku vykreslíme pomocí příkazu g.drawLine(int x1, int y1, int x2, int y2), kde x1 a y1 jsou souřadnice počátečního uzlu a x2, y2 jsou souřadnice koncového uzlu. Dále musíme přiřadit nakreslené hraně ohodnocení. To provedeme opět příkazem g.drawChars(char[] název, int offset, int délka, int x, int y). U každé hrany máme spočítanou souřadnici středu, kterou použijeme pro výpis ohodnocení. Tloušťku úsečky, ani její styl (přerušovaná, čárkovaná, …) v Javě nemůžeme nijak nastavit. Při výpisu minimální kostry grafu proto změníme barvu úsečky a danou úsečku vykreslíme hned čtyřikrát (Obrázek 5.1.2). Tím zajistíme přibližně stejnou šířku čáry, nezávisle na poloze počátečního a koncového uzlu.



Obrázek 5.1.2: Kreslení tučné hrany v Javě

5.1.4. Aplet pro výpočet min. kostry grafu pomocí Kruskalova algoritmu

Algoritmus pracuje s polem uzlů o velikosti *n*, kde *n* je celkový počet uzlů v grafu a s polem hran o velikosti *m*, kde *m* je celkový počet hran. Protože pole hran obsahuje všechny hrany grafu v pořadí jejich zadání, musíme nejprve pole setřídit. V apletech pracujeme s malými poli, neboť jsme omezeni maximálním počtem uzlů, které můžeme zadat (viz. 5.1.2). Proto bylo použito nejjednoduššího řazení, tzv. vkládání s výběrem nejmenšího prvku. Toto vkládání v prvním kroku nalezne minimum od *0* do *m-1* prvku a zařadí jej na *0-tou* pozici. V druhém kroku se hledá minimum od *1* do *m-1* prku a nalezený minimální prvek se zařadí na *1* pozici. Třídění končí po *m-1* krocích. Řazení výběrem nejmenšího prvku je přirozený stabilní algoritmus s celkovým počtem porovnání $O(n^2)$. Takto setříděné pole hran vstupuje do rekurzivní funkce realizující Kruskalův algoritmus. Zaveďme si výstupní pole hran, které bude obsahovat pouze hrany patřící do minimální kostry grafu. Víme, že toto pole bude mít velikost *n-1*, neboť celkový počet uzlů grafu je *n*. První hranu v poli setříděných hran hned přidáme do výstupního pole. Pak v cyklu od *1* do *m-1* vždy zkoumáme jednu hranu z pole setříděných hran. Tato hrana je pomocí rekurzivní funkce zkontrolována a pokud netvoří s již zařazenými hranami do výstupního pole cyklus, je k nim přidána. Rekurzivní funkce prochází všechny hrany zařazené do výstupního pole a porovnává jejich koncové uzly s koncovými uzly zkoumané hrany. V tomto cyklu zkoumáme a zařazujeme hrany až do chvíle, kdy přidáme *n-1* hranu do výstupního pole. V tu chvíli cyklus ukončíme a vykreslíme minimální kostru grafu dle požadavků uživatele (buď celou kostru grafu, nebo krok po kroku). Vykreslování uzlů a hran bylo popsáno v předchozí kapitole (viz. 5.1.3).

5.1.5. Aplet pro výpočet min. kostry grafu pomocí Jarníkova algoritmu

Aplet opět pracuje se dvěma vstupními poli. Pole uzlů má velikost *n*, kde *n* je celkový počet uzlů a pole hran má velikost *m*, kde *m* je celkový počet hran. U Jarníkova algoritmu není třeba nijak uspořádávat vstupní pole. Zaveďme si opět výstupní pole pro hrany minimální kostry o velikosti *n-1*. Dále budeme potřebovat proměnnou, ve které bude uložen počáteční uzel výpočtu algoritmu. Tento počáteční uzel je zadán uživatelem před spuštěním výpočtu. Můžeme jej nazvat modrým stromem, který se v dalších krocích bude rozrůstat. Dále budeme potřebovat pole uzlů patřících do modrého stromu (MS). Do tohoto pole můžeme ihned přidat počáteční uzel zadaný uživatelem. Nyní již můžeme přistoupit k vlastní funkci realizující Jarníkův algoritmus. Jde o cyklus *do .. while*, ve kterém se nalezne ze všech uzlů umístěných

v poli uzlů modrého stromu (MS) minimální incidentní hrana a pokud netvoří cyklus s modrým stromem (nemá oba koncové uzly v modrém stromu), je přidána do výstupního pole. Cyklus *do .. while* končí při umístění *n-1* hrany do výstupního pole hran. Pak opět následuje vykreslovací rutina, která je popsána v kapitole 5.1.3.

5.1.6. Aplet pro výpočet min. kostry grafu pomocí Borůvkova algoritmu

V tomto apletu bylo opět použito vstupní pole uzlů, hran a výstupní pole obsahující hrany minimální kostry grafu. Velikosti použitých polí jsou stejné, jako v kapitole 5.1.4 a 5.1.5. Na počátku je každý uzel grafu považován za samostatný modrý strom. Každý uzel má ve své proměnné komponenta uvedeno číslo modrého stromu, který tvoří. Tato čísla jsou uzlům přidělena dle pořadí jejích zadání (tzn. neexistují žádné dva či více uzlů, které by měly stejné číslo ve své proměnné komponenta). Funkce realizující Borůvkův algoritmus je naprogramována rekurzivně a v každém svém kroku vykonává hned několik operací. Nejprve je zapotřebí nalézt ke každému modrému stromu incidentní hranu s minimálním ohodnocením (informaci o modrém stromu nese proměnná komponenta). Z množiny všech nalezených minimálních hran nyní musíme vybrat duplicitní hrany, tzn. hrany se stejnými koncovými uzly. Nyní již máme množinu jednoznačných hran, které můžeme přidat do výstupního pole. Nakonec nesmíme zapomenout sloučit staré modré stromy v nové a přečíslovat proměnnou komponenta u všech uzlů, které se sloučily s jiným modrým stromem. Tento postup opakujeme rekurzivně, dokud nemáme pouze jeden modrý strom. To poznáme, že u všech uzlů v proměnné komponenta bude stejné číslo. Pak již nám nic nebrání ve výpisu minimální kostry grafu dle uživatelských požadavků.

5.2. Aplikace

Aplikace jsou programovány také v JBuilderu 8 Personal. Tyto aplikace nemohly být realizovány pomocí apletů, protože aplety mají vysoké zabezpečení a při práci se vstupem a výstupem prostřednictvím datových souborů dochází k nepřekonatelným problémům. Pro práci se soubory se tedy v Javě hodí především aplikace.

5.2.1. Návrh aplikací

Ve všech naprogramovaných aplikacích se budeme snažit maximálně minimalizovat grafické prvky a zaměříme se na jejich jádra – algoritmy. Ve výše uvedených apletech bylo zvykem zadávání vstupních dat prostřednictvím klávesnice a myši. U aplikací tomu bude

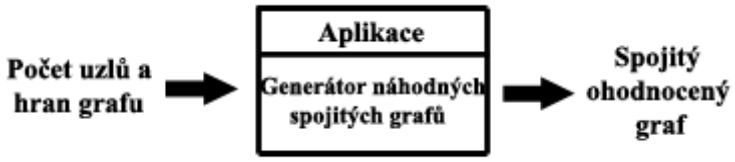
jinak. Převážná většina vstupů do programu bude prostřednictvím textových souborů a jen výjimečně se setkáme se vstupem z klávesnice (pomineme-li spouštění výpočtu).

5.2.2. Ovládání aplikací

Všechny navržené aplikace se ovládají intuitivně pomocí jednoho spouštěcího tlačítka a v případě náhodného generátoru spojitých grafů je toto tlačítko doplněno o dvě textová pole. Všechny navržené aplikace pracují s textovými soubory, které se ukládají resp. načítají z pracovního adresáře. Ten je pro všechny aplikace nastaven do adresáře, z kterého se aplikace spouští.

5.2.3. Aplikace pro náhodné generování spojitých ohodnocených grafů

Úkolem této aplikace je generovat náhodné spojité ohodnocené grafy a ukládat je v řádném formátu do textových souborů. Celá aplikace se skládá z jednoho tlačítka a dvou textových polí. Do textových polí uživatel zadá požadovaný počet uzlů a hran grafu a následným stisknutím tlačítka start spustí náhodné generování grafových hran. Při nekorektním zadání a spuštění aplikace je uživatel upozorněn a zadání musí opakovat. Poznamenejme, že za nekorektním zadáním může být písmeno, záporné číslo, desetinné číslo a nebo překročení maximálního možného počtu hran na zadaný počet uzlů (např. 4 uzly lze spojit maximálně 6 hranami).



Obrázek 5.2.1: Schéma vstupů a výstupů aplikace pro generování náhodných spojitých ohodnocených grafů

Nyní se podívejme na celý problém detailněji. Zavedeme si pole ohodnocení hran o velikosti m, kde m je celkový počet hran vytvářeného grafu. Nechť každá položka tohoto jednorozměrného pole obsahuje svůj index, tzn. že v tomto poli nenalezneme dvě položky se stejnou hodnotou. Máme-li vytvořit spojitý graf, musíme zajistit, aby všechny uzly grafu byly propojeny nějakou náhodnou hranou. Jeden z mnoha způsobů, jak to provést, je následující. První uzel propojíme s druhým, druhý s třetím atd. Toto propojení je realizováno v cyklu, kde

v každém kroku náhodně generujeme index pole ohodnocení hran a aktuální hranu ohodnotíme číslem uloženým na vygenerovaném indexu. Toto číslo již víckrát nesmíme použít, abychom generovaný graf mohli použít i pro Borůvkův algoritmus (Borůvkův algoritmus nemůže zpracovávat graf s dvěmi a více shodně ohodnocenými hranami). Proto na pozici vygenerovaného indexu zkopírujeme poslední položku tohoto pole a v příštím kroku náhodně generujeme v rozmezí o 1 menším. Po skončení tohoto cyklu postupujeme obdobně. Tentokrát již generujeme počáteční i koncový uzel současně s indexem ohodnocení hran. Každou náhodně vygenerovanou hranu musíme před umístěním do grafu zkontrolovat, zda-li tam již není s jiným ohodnocením (může se nám stát, že náhodně vygenerovaná hrana je již obsažena v grafu). Pokud ano, nepřidáváme ji ke grafu a znovu generujeme počáteční a koncový uzel.

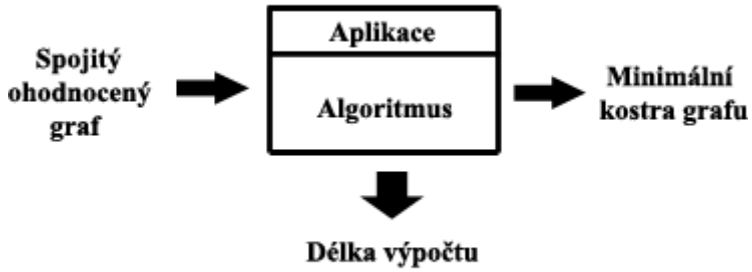
Nyní jsme se dostali do konečného stádia. Máme vygenerovaný náhodný spojitý ohodnocený graf, který je uložený v trojrozměrném poli. Každá položka tohoto pole nese informaci o počátečním a koncovém uzlu hrany a o jejím ohodnocení. V podobné struktuře se daný graf ukládá do textového souboru. V tomto textovém souboru je na prvním řádku tzv. hlavička, která obsahuje počet uzlů a hran grafu. Od druhého řádku jsou ukládány jednotlivé hrany grafu, každá na samostatném řádku. Hrana je reprezentována třemi číselnými hodnotami oddělenými mezerou. První je číslo počátečního uzlu, pak za mezerou následuje číslo koncového uzlu hrany a za další mezerou je ohodnocení hrany. Tyto textové soubory jsou snadno čitelné a snadno upravitelné.

Ukázka struktury dat v textovém souboru odpovídá s grafem na obrázku 2.4.1:

```
6 7
1 2 1
6 5 5
2 5 4
5 4 6
2 4 7
3 2 2
3 4 3
```

5.2.4. Aplikace pro měření časové náročnosti Kruskalova algoritmu

Hlavním úkolem této aplikace je měření celkového času potřebného k výpočtu Kruskalova algoritmu. Celý program se ovládá jedním tlačítkem start, které spouští algoritmus. Po spuštění jsou zobrazeny nejdůležitější informace o zpracovávaném grafu mezi něž patří počet uzlů a hran. Po zdárném ukončení výpočtu je zobrazen celkový čas potřebný ke zpracování zadaného úkolu.



Obrázek 5.2.2: Schéma vstupů a výstupů aplikací, které měří časové náročnosti algoritmů

Algoritmus při svém spuštění otevře textový soubor a načte celý graf do třírozměrného pole hran. Kruskalův algoritmus pracuje se setříděným polem hran dle jejich ohodnocení. Optimální náročnost Kruskalova algoritmu je $O(m \log m)$. Chceme-li naprogramovat Kruskalův algoritmus optimálně, tj. s časovou náročností $O(m \log m)$, nesmíme použít algoritmus řazení s vyšší časovou náročností. V této aplikaci bylo použito řazení QuickSort, jehož časová náročnost je skoro jistě $O(m \log m)$. Protože v tříděném poli má každá hrana jiné ohodnocení, použil jsem QuickSortu, který dělí pole na dvě části (pokud bychom měli více prvků se stejnou hodnotou, je vhodné pole rozdělit na tři části). Celý algoritmus třídění je založen na principu, že nejefektivnější výměny jsou na velké vzdálenosti. Proto se náhodně vybere index k z pole (v našem případě je to prostřední prvek), který nám rozdělí pole na dvě části. Při přerovnávání postupujeme od konců proti sobě a prvky porovnáváme s pivotem (hodnota na indexu k) a prohazujeme je. Když se nám indexy proti sobě běžících cyklů střetnou, voláme QuickSort rekurzivně pro každou podčást.

Pro výpočet Kruskalova algoritmu si budeme muset opět zavést několik polí a tříd. Opět potřebujeme nějaké výstupní pole hran o velikosti *n-1* (*n* je počet uzlů grafu), do kterého

budeme ukládat hrany patřící do minimální kostry grafu. Dále si zaveďme jednorozměrné pole *S* o velikosti *n*, které ponese informaci o příslušnosti jednotlivých uzlů k modrým stromům. Pod indexem si představme uzel a pod hodnotou číslo stromu, ve kterém daný uzel leží. Na počátku každou položku pole vyplníme dle jejího indexu. Dále si zavedeme třídu *Strom* obsahující proměnné *délka* a *list* a třídu *List* s proměnnými *uzel* a *další*. Vytvoříme pole *Stromů* a *Listů* o velikosti *n*. Při inicializaci naplníme proměnné těchto polí následovně:

S

uzel	*strom*
0	0
1	1
2	2
:	:
n-1	n-1

Strom

strom	*délka*	*konec*
0	1	0
1	1	1
2	1	2
:	:	:
n-1	1	n-1

List

list	*uzel*	*další*
0	0	0
1	1	0
2	2	0
:	:	:
n-1	n-1	0

Celý výpočet Kruskalova algoritmu probíhá v cyklu, ve kterém procházíme všechny setříděné hrany. Nyní si popíšeme jeden běh tohoto cyklu, který rozhodne o zařazení právě jedné hrany do minimální kostry grafu. Nejprve musíme u zkoumané hrany ověřit, zda-li oba její koncové uzly neleží ve stejném modrém stromu (hrana nesmí tvořit s modře obarvenými hranami kružnici). Pokud je tomu tak, můžeme hranu zařadit do výstupního pole, neboť zcela jistě bude součástí minimální kostry grafu. Přidaná hrana nám propojila nějaké dva modré stromy a my je nyní potřebujeme přečíslovat. Vždy přečíslujeme strom s menším počtem uzlů. Informace o velikosti jednotlivých stromů jsou uloženy v proměnných *délka* v poli všech stromů. Při slučování dvou stromů musíme zvýšit délku nepřeznačovaného stromu o délku přečíslovaného stromu, v poli *Listů* změníme ukazatele na další listy a nakonec upravíme v poli *Stromů* konec sloučených stromů. Hrany zařazujeme do výstupního pole až do okamžiku, kdy zadáme *n-1* hranu. Dále následuje zápis do souboru dle instrukcí uvedených v kapitole 5.2.3, doplněný o jednu hlavičkovou informaci. Protože hlavním úkolem aplikace je měření celkového času potřebného pro výpočet minimální kostry grafu, do hlavičkových informací výstupní minimální kostry grafu je zapsán naměřený čas v ms (hlavička výstupního textového souboru : počet uzlů, počet hran, naměřený čas v ms). Popis měření časové náročnosti algoritmu je uveden v kapitole 6.1.

5.2.5. Aplikace pro měření časové náročnosti Jarníkova algoritmu

Ovládání této aplikace je totožné s ovládáním aplikace pro měření časové náročnosti algoritmů pomocí Kruskalova algoritmu (viz. 5.2.4). Schéma vstupů a výstupů je patrné z obrázku 5.2.2.

Jarníkův algoritmus pracuje s časovou náročností $O(n^2)$. Proto se nám přímo nabízí možnost použití maticové reprezentace grafu, která je velice přehledná a srozumitelná, neboť vyplnění takové matice nám zabere právě $O(n^2)$. Na počátku celého výpočtu opět budeme muset načíst zkoumaný graf z textového souboru do třírozměrného pole a vytvoříme si výstupní pole pro hrany minimální kostry grafu (viz. 5.2.4). Zaveďme si novou třídu *Hrana*, která bude obsahovat proměnné *počátek*, *konec* a *ohodnocení*. Dále budeme potřebovat matici *Hran* o rozměru $n \times n$. Vytvoříme proto pole hran s názvem *Matice* o rozměru $n \times n$. Nyní projdeme všechny hrany ve vstupním poli a každou hranu zařadíme do matice. Řádek a sloupec matice nechť představují počáteční a konečný uzel hrany. Na tuto pozici a pozici symetrickou dle hlavní diagonály uložíme hranu do proměnných *počátek*, *konec* a *ohodnocení*. Takto vytvořená matice je symetrická dle hlavní diagonály (počáteční uzel může být koncovým a naopak). Ukažme si zápis jednoduchého spojitého ohodnoceného grafu (viz. Obrázek 2.4.1) v maticové podobě:

	1	2	3	4	5	6
1	X	1, 2, 1				
2	1, 2, 1	X	2, 3, 2	2, 4, 7	2, 5, 4	
3		2, 3, 2	X	3, 4, 3		
4		2, 4, 7	3, 4, 3	X	4, 5, 6	
5		2, 5, 4		4, 5, 6	X	5, 6, 5
6					5, 6, 5	X

V každé buňce matice jsou uloženy základní informace o hraně v následujícím pořadí:

(*počátek, konec, ohodnocení*)

Hlavní diagonála je prázdná, neboť graf neobsahuje smyčky.

V každém sloupci a řádku matice jsou uvedeny incidentní hrany k totožnému uzlu s indexem řádku či sloupce. V každém kroku si zvolme za výchozí uzel pro výpočet Jarníkova algoritmu uzel s maximálním číslem (poslední sloupec či řádek matice). Z tohoto uzlu najdeme incidentní hranu s minimálním ohodnocením. Tuto hranu přidáme do výstupního pole hran minimální kostry. Nyní nám zbývá sloučit koncové uzly přidané hrany v jeden modrý strom. Vždy strom s nižším číslem zůstává a strom s vyšším číslem se k němu připojuje.

Porovnáváme prvky se stejnými řádkovými indexy a vždy hranu s nižším ohodnocením zapíšeme na pozici menšího sloupcového indexu. V tuto chvíli již na nic poslední sloupec a řádek matice nepoužijeme a proto v dalším cyklu algoritmu pracujeme s maticí o řádek a sloupec menší.

Algoritmus uvedený v minulém odstavci je jistou modifikací Jarníkova algoritmu. V klasickém Jarníkově algoritmu se vždy rozrůstá jeden modrý strom, ale v našem případě necháme vždy rozrůstat strom v posledním sloupci či řádku matice. Takováto implementace je o něco jednodušší a rychlejší než klasický Jarníkův algoritmus. Kdybychom však chtěli dodržet vlastnost, že se vždy rozrůstá jeden modrý strom, stačilo by na konci každého našeho Jarníkova kroku prohodit právě sloučené dva stromy s předposledním sloupcem a řádkem matice. Tím bychom aktuálně sloučené stromy umístili na poslední aktivní pozice matice a v dalším kroku by se opět hledala incidentní hrana k tomuto modrému stromu.

Dále následuje pouze zápis výsledné minimální kostry grafu do textového souboru. Podrobněji je to popsáno v předchozí kapitole 5.2.4.

5.2.6. Aplikace pro měření časové náročnosti Borůvkova algoritmu

Aplikace pro měření časové náročnosti Borůvkova algoritmu se ovládá tlačítkem start, po jehož stisknutí se z textového souboru načte požadovaný graf a zobrazí se jeho celkový počet uzlů a hran. Po ukončení výpočtu je zobrazen celkový čas potřebný ke zpracování zadaného úkolu. Schéma vstupů a výstupů této aplikace je uvedeno na obrázku 5.2.2.

Borůvkův algoritmus pracuje s časovou náročností $O(m \log n)$, a proto nemůžeme při výpočtu použít matic. Stejně jako u Jarníkova algoritmu si zavedeme několik pomocných tříd a polí. Zaveďme si třídu *Hrana*, která bude obsahovat proměnné *počátek*, *konec*, *ohodnocení*, *nový_počátek* a *nový_konec*. Dále třídu *Listy* o proměnných *číslo* a *další* a třídu *Inc* s proměnnými *počátek* a *konec*. Opět potřebujeme vstupní pole všech hran vyšetřovaného grafu a výstupní pole hran patřících do minimální kostry grafu. Dále si zavedeme pole *Hran Hrana* a *Nová_hrana* o velikosti celkového počtu hran *m*. V poli *Hrana* jsou uloženy všechny hrany vstupního grafu. Dále si zavedeme pole *minimálních_hran*, *jednoznačných_minimálních_hran* a pole *S*. Tato pole mají shodnou délku *n*, kde *n* je celkový počet uzlů. Nakonec zavedeme pole *Inc* o velikosti *n* a pole *Listů Listy* o velikosti $3 \cdot n - 2$. Jednotli-

vé významy polí si uvedeme při popisu algoritmu. Ukažme si datové struktury, s kterými budeme pracovat po inicializaci.

Hrana,Nová_hrana (min. a min.jednoznačná hrana-velikost n)

	počátek	*konec*	*ohodnocení*	*nový_počátek*	*nový_konec*
0	-1	-1	-1	-1	-1
1	-1	-1	-1	-1	-1
2	-1	-1	-1	-1	-1
:	:	:	:	:	:
m-1	-1	-1	-1	-1	-1

Listy

	číslo	*další*
0	-1	-1
1	-1	-1
2	-1	-1
:	:	:
3.n-2	-1	-1

Inc

	počátek	*konec*
0	-1	-1
1	-1	-1
2	-1	-1
:	:	:
n-1	-1	-1

S

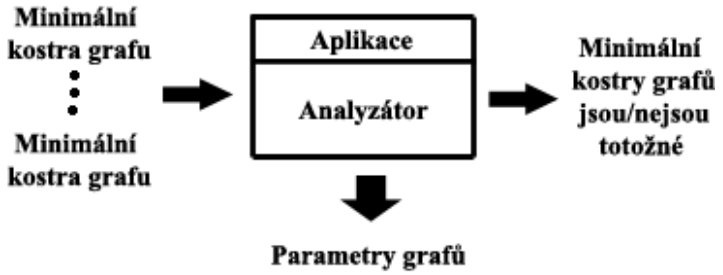
0	-1
1	-1
2	-1
:	:
n-1	-1

Nyní přistupme k vlastní realizaci Borůvkova algoritmu. Na následujících řádcích si popíšeme činnost jednoho kroku Borůvkova algoritmu. Na počátku každý uzel grafu představuje samostatný modrý strom. Musíme tedy nalézt z každého uzlu grafu incidentní hranu s minimálním ohodnocením. Takto vybrané hrany zapíšeme do pole *minimálních_hran*. Toto pole však může obsahovat duplicitní hrany, tzn. hrany vybrané oběma svými koncovými uzly. Proto do pole *jednoznačných_minimálních_hran* zapíšeme pouze takové hrany, které nezpůsobí v tomto poli duplicitu. Všechny hrany zapsané v tomto poli můžeme zapsat do výstupního pole, které obsahuje hrany minimální kostry grafu. Nyní naplníme pole incidence *Inc* a pole *Listů* ukazateli na společné listy modrých stromů sloučených pomocí hran v poli *jednoznačných_minimálních_hran*. Voláme rekurzivní proceduru *Přepiš* dvou vstupních parametrů. Prvním parametrem je číslo starého stromu a druhým číslo nového stromu. Funkce sloučí dva modré stromy, přičemž přepíše starý modrý strom na nový. Tuto rekurzivní funkci

voláme pro každý starý strom, tj. pro každý uzel grafu zpracovávaného v daném kroku Borůvkova algoritmu. Výstupem této funkce je vyplněné pole *S*. Každý jeho index si můžeme představit jako uzel. Hodnota u každého uzlu (indexu) vyjadřuje příslušnost daného uzlu k modrému stromu. Když máme vytvořené pole *S*, můžeme přistoupit k přečíslování *nových_začátků* a *nových_konců* u hran uložených v poli *Hran*. Vytvoříme nový redukovaný graf obsahující všechny hrany z pole *Hrana*, které nemají *nový_počátek* a *nový_konec* ve stejném modrém stromu. Tento nový redukovaný graf uložíme do pole *nová_hrana*. Nakonec tohoto Borůvkova kroku přepíšeme pole *hrana* polem *nová_hrana* a aktualizujeme počet hran a uzlů nového redukovaného grafu. Takto popsaný Borůvkův krok opakujeme, dokud počet modrých stromů je větší než 1.

5.2.7. Aplikace pro porovnání vypočtených minimálních koster grafu

Výše zmíněné algoritmy nalezly minimální kostru zadaného grafu a uložily ji do textového souboru. Každý z algoritmů pracuje s jinou časovou náročností, ale výsledné minimální kostry stejného vstupního grafu musí být vždy stejné. Pokud by nebyly, znamenalo by to chybně naprogramovaný algoritmus, nebo nekorektně zadaný vstupní graf. Pokud jsou vstupní grafy dostatečně malé, správnost nalezené minimální kostry se dá lehce určit. Avšak při velkých vstupních grafech nám v tom může napomoci právě tato aplikace.



Obrázek 5.2.3: Schéma vstupů a výstupů aplikace pro porovnání vypočtených minimálních koster grafu

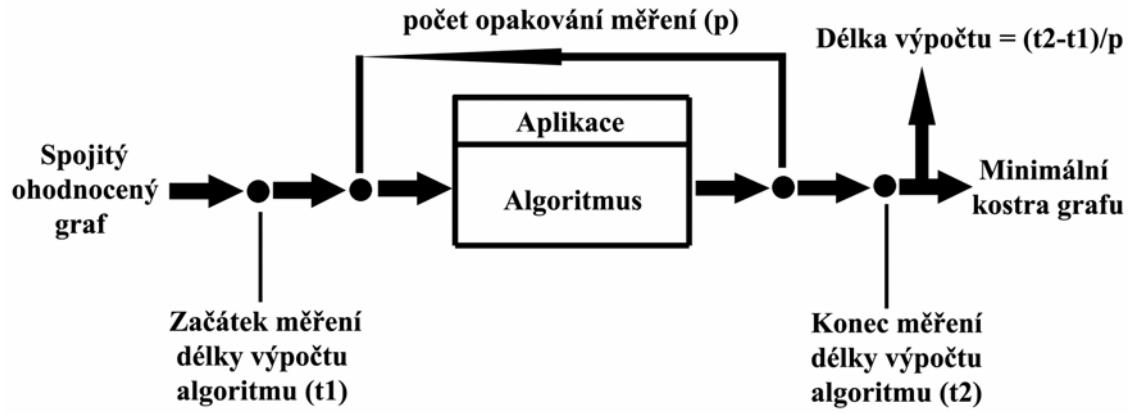
Aplikace načte tři vstupní soubory obsahující minimální kostru grafu spočtenou pomocí Kruskalova, Jarníkova a Borůvkova algoritmu. Připomeňme si, že minimální kostra získaná Kruskalovým algoritmem má všechny hrany uspořádány dle svého ohodnocení, na rozdíl od

Borůvkova a Jarníkova algoritmu, kde uspořádání hran je náhodné. Po načtení všech tří grafů nejprve ověříme, zda-li u všech tří grafů souhlasí počty uzlů a hran. Pokud ano, přistoupíme k druhé části verifikace, kde uspořádáme hrany dle jejich ohodnocení a porovnáme hrany všech tří souborů. Hrany uspořádáme pomocí QuickSortu (viz. 5.2.4). Pokud se hrany na totožných indexech polí nerovnají, program oznámí uživateli, že porovnávané minimální kostry grafů nejsou totožné.

6. Výsledky měření časových náročností algoritmů

6.1. Měření délky výpočtu v Javě

[1] Program v Javě běží pod konkrétním operačním systémem, jehož prostředí je popsáno v systémových atributech (*system attributes*). Pod těmi si můžeme představit např. jméno operačního systému, domovský adresář uživatele atd. Java umožňuje čtení těchto atributů a tím výrazně napomáhá k vytváření přenositelných programů. Systémově nezávislé atributy jsou přístupné pomocí třídy *System*. Tato třída poskytuje metodu *currentTimeMillis()*, která vrací počet milisekund od 1.1.1970. Nás bude zajímat rozdíl takovýchto dvou hodnot, kterým je celkový čas běhu programu. Na následujícím obrázku 6.1.1 je uvedeno schéma měření časových náročností algoritmů.



Obrázek 6.1.1: Schéma měření časových náročností algoritmů

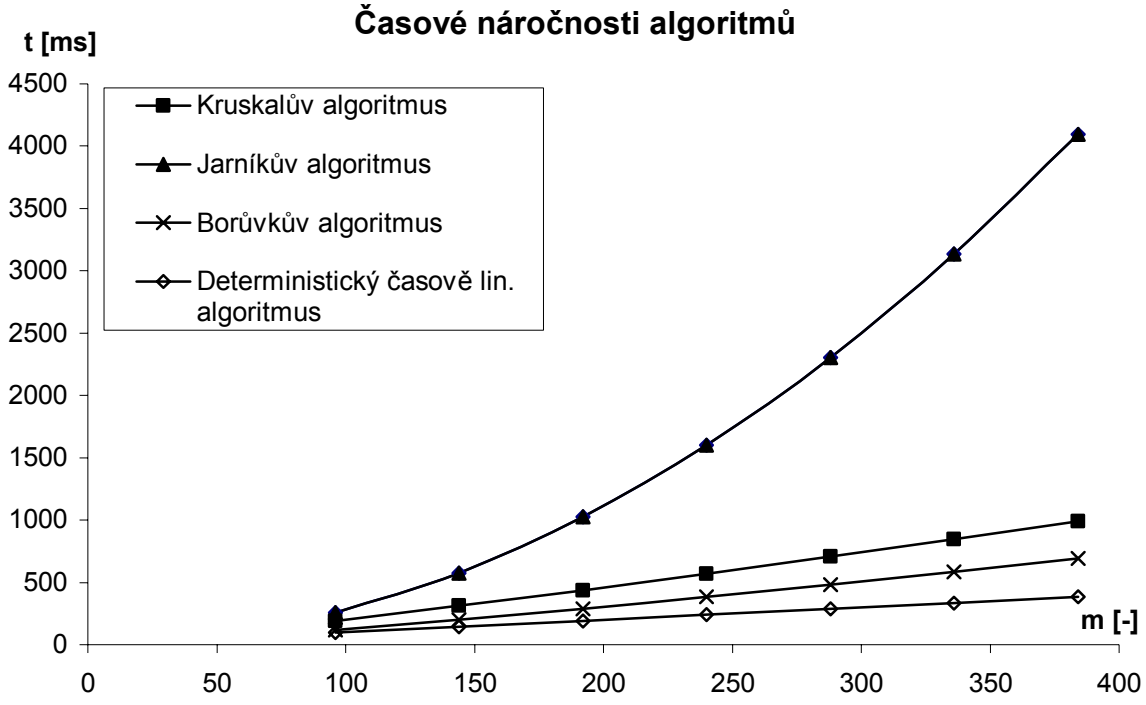
Musíme si uvědomit, že moderní operační systémy jsou víceúlohové a víceuživatelské. Měřený program proto nikdy neběží sám a z tohoto důvodu se mohou doby běhu samotného programu značně lišit. Před měřením časových závislostí jednotlivých algoritmů je vhodné uzavřít všechny nepotřebné spuštěné aplikace. Některé se nám nemusí podařit uzavřít, a proto se je snažíme při výpočtu neaktivovat. Abychom dosáhli co možná nejpřesnějších výsledků, spustíme měřený program hned několikrát, vyloučíme extrémy a ze zbylých hodnot spočteme průměr.

6.2. Porovnání časových náročností algoritmů

V této diplomové práci byly naprogramovány tři klasické metody řešení minimální kostry grafu. Každá z těchto metod pracuje s jinou časovou náročností a hodí se pro výpočet jiných typů grafů. Tabulka 6.2.1 uvádí jednotlivé časové náročnosti klasických a deterministického časově lineárního algoritmu, Graf 6.2.1 zobrazuje jejich časové závislosti.

Název algoritmu	Časová náročnost
Kruskalův algoritmus	O(m.log m)
Jarníkův algoritmus	$O(n^2)$
Borůvkův algoritmus	O(m.log n)
Deterministický časově lineární algoritmus	O(m)

Tabulka 6.2.1: Časové náročnosti algoritmů



Graf 6.2.1: Závislosti časových náročností algoritmů

Málokdy se setkáme s úkoly počítat minimální kostry z plných grafů, tzn. grafů, které mají *n* uzlů a *n.(n-1)/2* hran. Mnohem častěji se setkáme s „reálnými“ grafy, jejichž počet

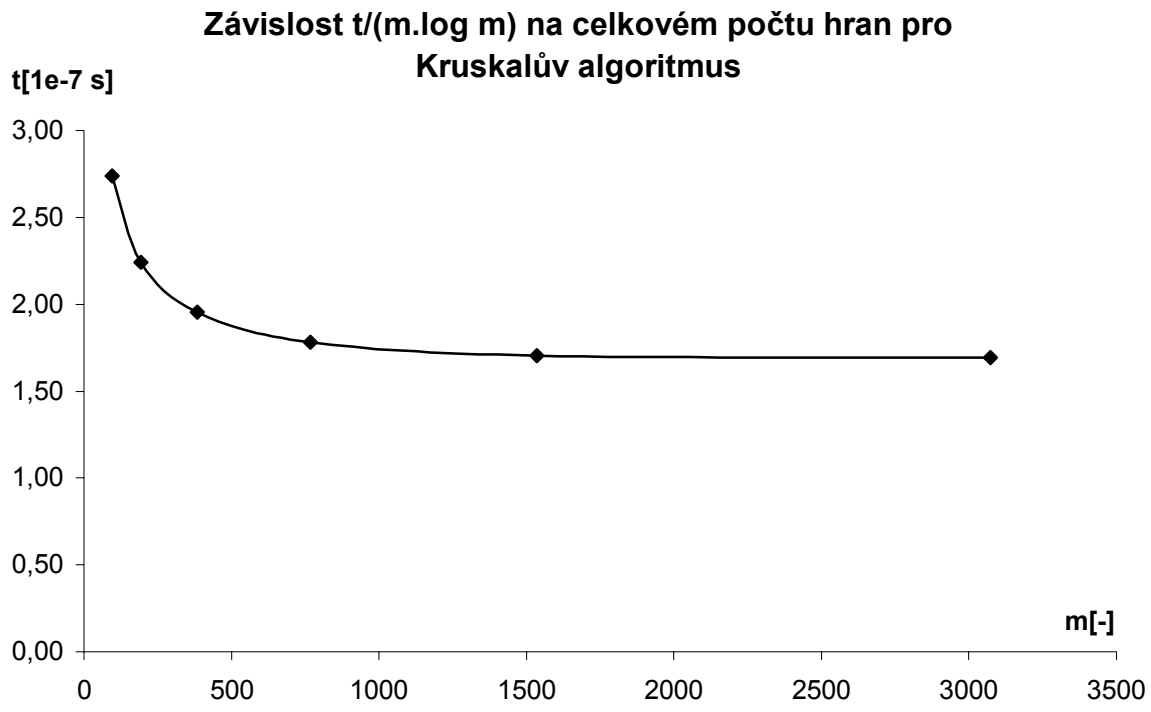
hran je nějakým malým násobkem počtu uzlů. Proto byly náhodně vygenerovány grafy s *n* uzly a *m=6*n* hranami. Graf 6.2.1 popisuje časové závislosti grafů s předem definovanými vlastnostmi.

V následujících tabulkách 6.2.2, 6.2.3, 6.2.4 se setkáme s naměřenými časovými hodnotami jednotlivých algoritmů. Dále jsou pro každý algoritmus uvedeny dva grafy. První popisuje závislost poměru naměřeného času a obecné náročnosti algoritmu na celkovém počtu hran. Výsledná křivka musí ležet pod nějakou konstantou, tzn., že tato křivka nesmí překročit nějakou konstantní hodnotu (viz. Graf 6.2.2, Graf 6.2.4, Graf 6.2.6). Druhý graf zobrazuje výslednou časovou náročnost algoritmu. První křivka v tomto obrázku značí naměřenou časovou náročnost algoritmu. Druhá křivka znázorňuje obecnou časovou náročnost algoritmu přepočítanou k první naměřené hodnotě, tzn. že spočítáme poměr naměřeného času prvního (nejmenšího) grafu s obecnou časovou náročností tohoto grafu (př. pro Kruskalův algoritmus : k=t/m.log m). Tímto podílem získáme časovou konstantu, se kterou pak násobíme každou obecnou časovou náročnost (př. pro Kruskalův algoritmus náročnost$_i$=náročnost$_i$.k), kde i=1..počet řešených grafů). Křivka naměřené časové náročnosti nesmí v žádném bodě překročit obecnou časovou náročnost přepočítanou k prvnímu (nejmenšímu) měřenému grafu (viz. Graf 6.2.3, Graf 6.2.5, Graf 6.2.7).

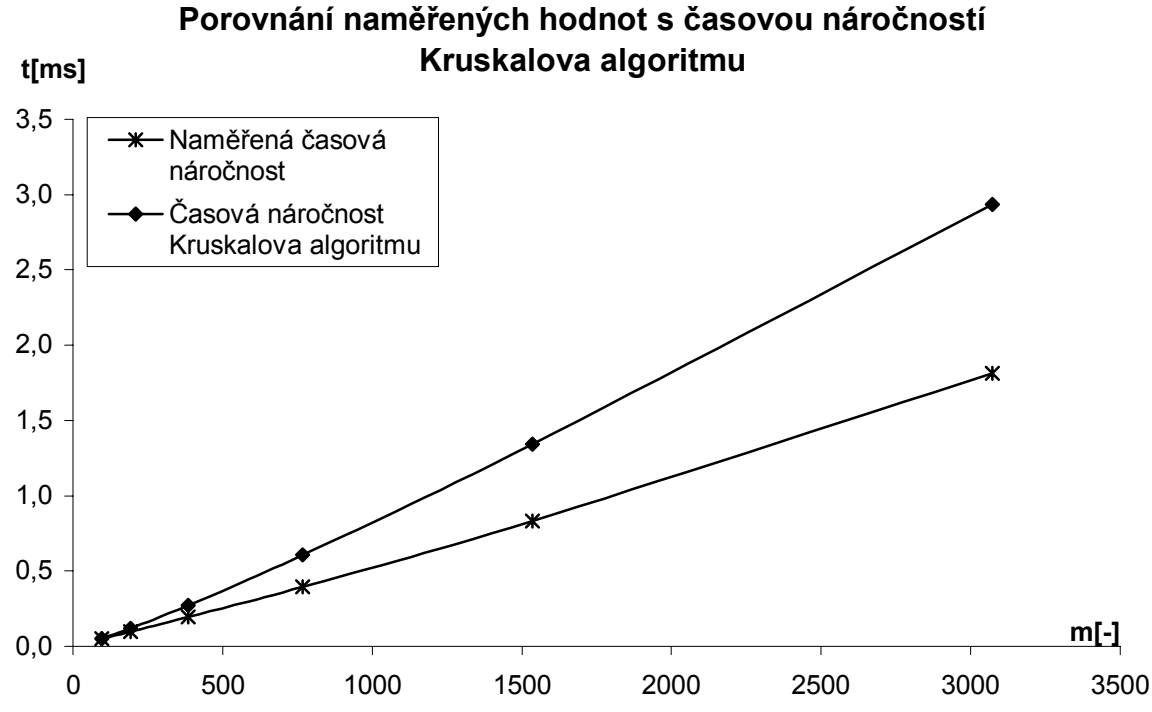
Při měření časové náročnosti Kruskalova algoritmu jsme pomocí náhodného generátoru (viz. 5.2.3) vygenerovali šest náhodných spojitých ohodnocených grafů. Výpočet minimální kostry grafu jsme pro každý graf opakovali 100 000 krát a ze získaných hodnot jsme spočetli aritmetický průměr. Všechny takto získané hodnoty jsou uvedeny v tabulce 6.2.2 a doplněny o obecné časové náročnosti a poměr *t/m.log m*. Graf 6.2.2 ukazuje, že poměr změřeného času a obecné náročnosti algoritmu má asymptotický charakter.

n[-]	m[-]	t[ms]	m.log m[-]	t/m.log m[ms]
16	96	0,05210	190,3	0,000274
32	192	0,09820	438,4	0,000224
64	384	0,19380	992,4	0,000195
128	768	0,39505	2216,0	0,000178
256	1536	0,83420	4894,3	0,000170
512	3072	1,81490	10713,4	0,000169

Tabulka 6.2.2: Naměřená časová náročnost pro Kruskalův algoritmus



Graf 6.2.2: Závislost *t/(m.log m)* na celkovém počtu hran pro Kruskalův algoritmus

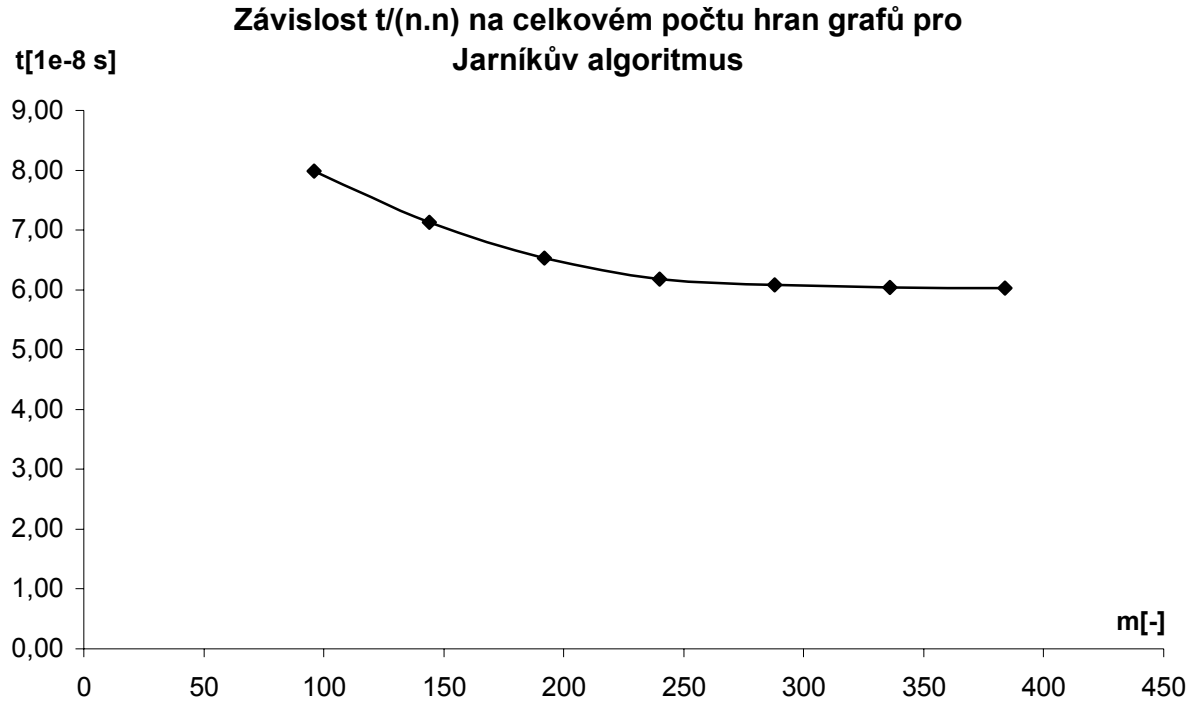


Graf 6.2.3: Porovnání naměřených hodnot s časovou náročností Kruskalova algoritmu

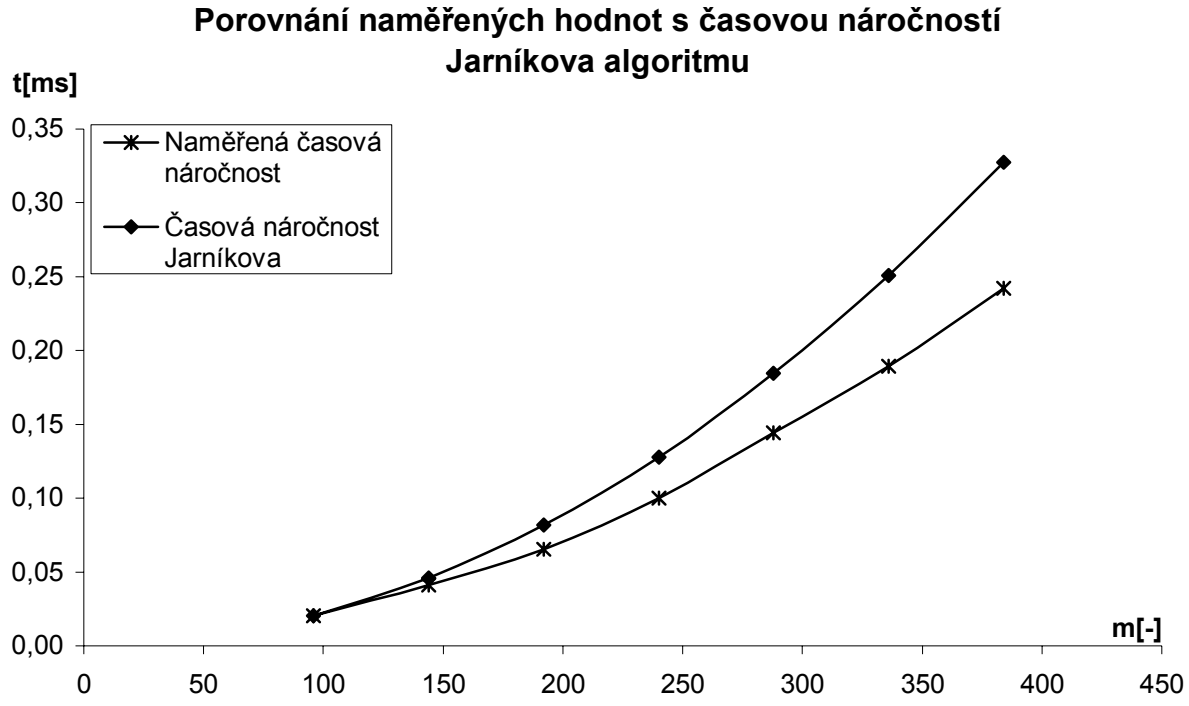
Pro změření časové náročnosti Jarníkova algoritmu bylo použito stejného postupu jako v případě minulém. Opět jsme použili náhodně vygenerované grafy, ale tentokrát nám postačilo k dokázání asymptotického ustálení hodnoty t/n^2 použití mnohem menších grafů.

n[-]	m[-]	t[ms]	n^2[-]	t/n^2[ms]
16	96	0,02043	256	0,000080
24	144	0,04106	576	0,000071
32	192	0,06690	1024	0,000065
40	240	0,10004	1600	0,000062
48	288	0,14010	2304	0,000061
56	336	0,18927	3136	0,000060
64	384	0,24703	4096	0,000060

Tabulka 6.2.3: Naměřená časová náročnost pro Jarníkův algoritmus



Graf 6.2.4: Závislost t/n^2 na celkovém počtu hran pro Jarníkův algoritmus

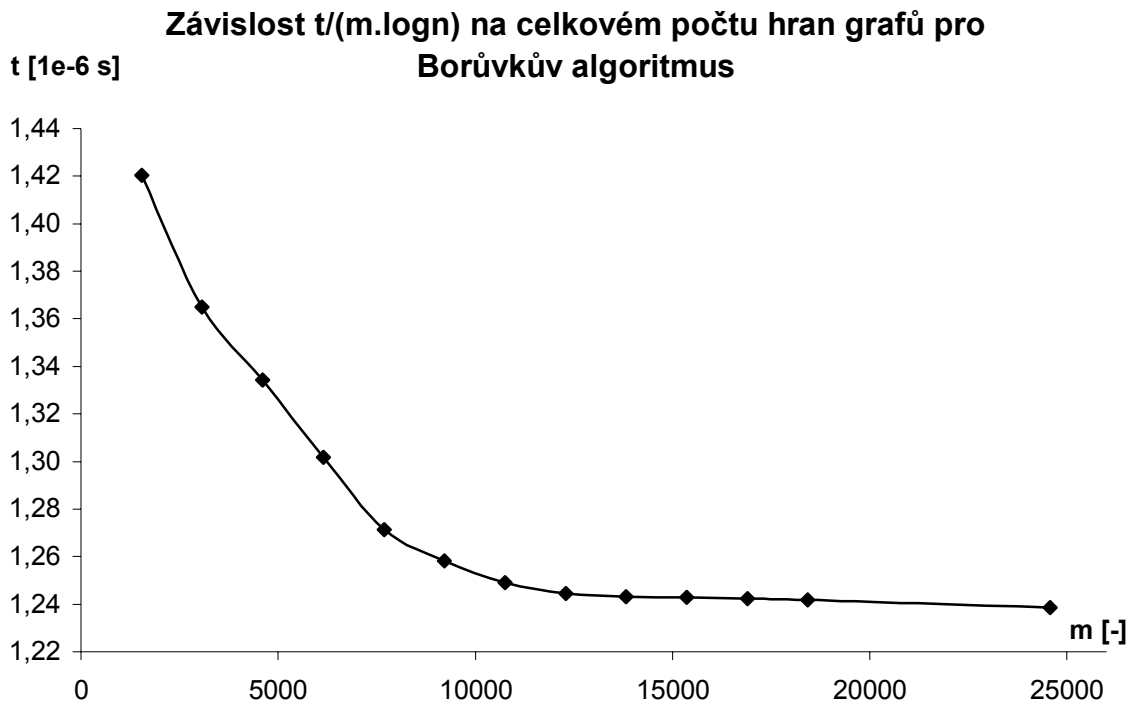


Graf 6.2.5: Porovnání naměřených hodnot s časovou náročností Jarníkova algoritmu

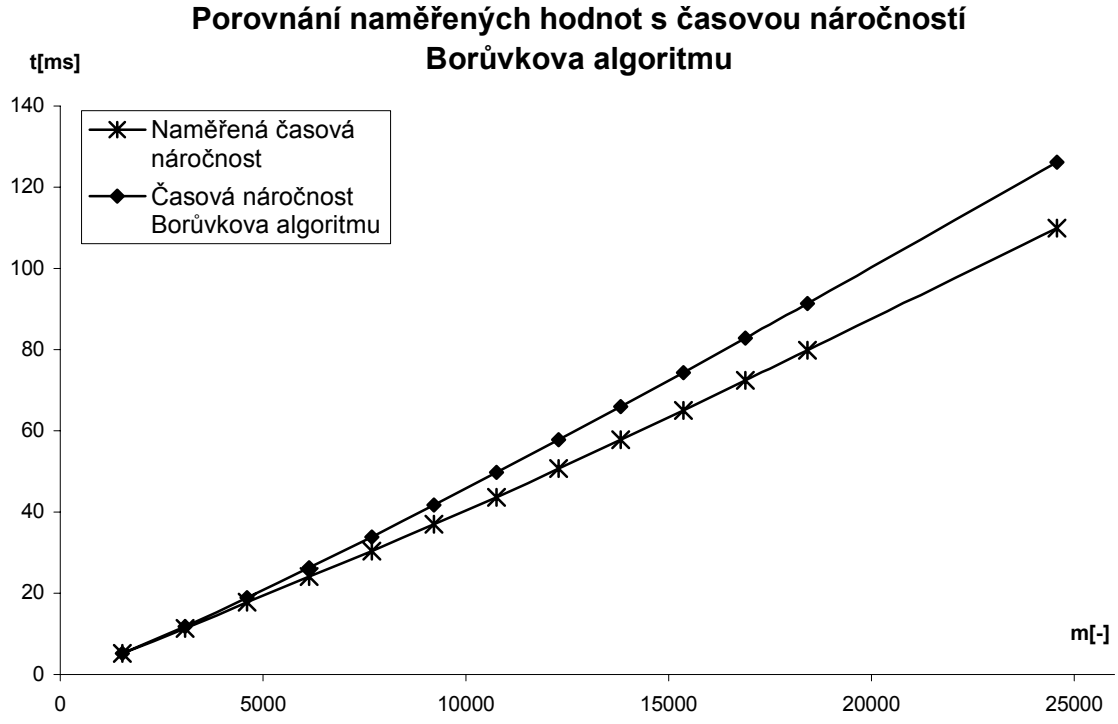
Časová náročnost Borůvkova algoritmu byla měřena na 13 poměrně rozsáhlých spojitých grafech. Výpočet jsme tentokrát pro každý graf opakovali pouze 2 000 krát, protože výpočetní čas několikanásobných grafů oproti minulým dvěma případům je mnohem delší.

n [-]	m [-]	t [ms]	m.log n[-]	t/m.log n[ms]
256	1536	5,254	3699,05658	0,0014204
512	3072	11,361	8322,87732	0,0013650
768	4608	17,742	13295,74451	0,0013344
1024	6144	24,074	18495,28293	0,0013016
1280	7680	30,340	23863,37257	0,0012714
1536	9216	36,950	29365,78144	0,0012583
1792	10752	43,690	34979,89023	0,0012490
2048	12288	50,632	40689,62245	0,0012443
2304	13824	57,780	46482,95773	0,0012430
2560	15360	65,060	52350,56587	0,0012428
2816	16896	72,410	58284,99326	0,0012423
3072	18432	79,820	64280,14777	0,0012418
4096	24576	109,960	88777,35808	0,0012386

Tabulka 6.2.4: Naměřená časová náročnost pro Borůvkův algoritmus



Graf 6.2.6: Závislost *t/(m.log n)* na celkovém počtu hran pro Borůvkův algoritmus



Graf 6.2.7: Porovnání naměřených hodnot s časovou náročností Jarníkova algoritmu

6.3. Zhodnocení dosažených výsledků

Z tabulek 6.2.2, 6.2.3, 6.2.4 je jasně vidět asymptotické chování hodnot *t/náročnost*. V případě Kruskalova algoritmu nám k tomu postačily středně velké grafy, u Jarníkova algoritmu nám stačily malé grafy a Borůvkův algoritmus si vyžádal velikých grafů (3072 uzlů a 18432 hran). Na grafech 6.2.2, 6.2.4, 6.2.6 je názorně vidět asymptotické chování křivek poměru naměřeného času a obecné časové náročnosti.

Aplikováním algoritmů na různě husté grafy naměříme rozdílné délky výpočtů. Zvolením vhodného algoritmu můžeme velice urychlit výpočet minimální kostry grafu.

Náročnost Kruskalova algoritmu je určená rychlostí použitého třídícího algoritmu, který uspořádá vzestupně hrany dle jejich ohodnocení. Proto se Kruskalův algoritmus hodí především na řídké grafy, které nemají příliš moc hran. Třídící algoritmus zpracovává menší počet hran, a proto Kruskalův algoritmus vypočte minimální kostru takového grafu mnohem rychleji, než zbývající dva klasické algoritmy.

Jarníkův algoritmus využívá při výpočtu minimální kostry grafu matic. Vyplnění a práce s maticí zabírá nejvíce výpočetního času tohoto algoritmu. Když jednou takovou matici vytvoříme, při práci s ní nám je jedno, jestli její pozice jsou z poloviny nebo zcela zaplněny. Pokud je taková matice zcela zaplněna, její zpracování je stejně náročné jako zpracování velmi řídce zaplněné matice. Proto se Jarníkův algoritmus hodí pro zpracování hustých grafů s velkým počtem hran (řádově n^2) a malým počtem uzlů. Má-li graf *n* uzlů, může mít maximálně $n \cdot (n-1)/2$ hran.

Borůvkův algoritmus je nejvýhodnější použít na „reálné“ středně zaplněné grafy. Pro řídké grafy dosahuje Borůvkův algoritmus horších výsledků než Kruskalův.

7. Závěr

Diplomová práce řeší problém hledání minimální kostry grafu ve spojitém ohodnoceném grafu. Byly naprogramovány tři výukové aplety a pět aplikací v moderním programovacím jazyce Java. Pro naprogramované aplety byly vytvořeny internetové stránky shrnující základní přístupy řešení problému minimální kostry grafu. V rámci této diplomové práce se také podařilo přeložit, opravit a doplnit verifikační algoritmus, který je základem nejdokonalejších algoritmů.

Naprogramované aplety slouží k hledání minimální kostry grafu prostřednictvím Kruskalova, Jarníkova a Borůvkova algoritmu. Umožňují uživateli vytvářet a modifikovat vlastnoručně nakreslené grafy a samozřejmě tyto grafy řeší.

Tyto tři aplety jsou implementovány do výukových internetových stránek. Stránky obsahují teoretické základy hledání minimálních koster grafů a názorné příklady jejich řešení. Prostřednictvím apletů si uživatel může ověřit nabyté teoretické znalosti.

Všechny vytvořené aplikace byly maximálně oproštěny od grafických prvků a soustředí se především na svoji rychlost. První aplikace řeší problém generování náhodných spojitých ohodnocených grafů. Další tři se zabývají měřením časových náročností klasických algoritmů, řešících problém minimální kostry grafu. U všech tří aplikací se podařilo dosáhnout předepsaných náročností. Poslední aplikace je zaměřena na kontrolu spočtených minimálních koster.

Realizované programové vybavení problému hledání minimální kostry grafu lze využít jako doprovodnou pomůcku při výuce základů diskrétní matematiky a operační analýzy na Technické univerzitě v Liberci.

Seznam použité literatury

[1] **Herout, Pavel**: Učebnice jazyka Java, Kopp 2001

[2] **Herout, Pavel**: Java – grafické uživatelské prostředí, Kopp 2001

[3] **Eckel, Bruce.**: Myslíme v jazyku Java, Grada Publishing 2000

[4] **Milková, Eva**: Moderní pohled na „jistý problém minimální“. Pokroky matematiky, fyziky a astronomie, ročník 45/2000

[5] **Milková, Eva**: Problém minimální kostry grafu, Gaudeamus, Hradec Králové 2001

[6] **Holenda J., Ryjáček Z.**: Lineární algebra II, ZČU 1997

[7] **Nešetřil J.**: Teorie grafů, SNTL 1979

[8] **David R. Karger, Philip N. Klein, Robert E. Tarjan**: A Randomized Linear-Time Algorithm to Find Minimum Spanning Trees, článek je přístupný na adrese: http://citeseer.nj.nec.com/karger95randomized.html

[9] **Valerie King**: A Simpler Minimum Spanning Tree Verification Algorithm, článek je přístupný na adrese: http://citeseer.nj.nec.com/king97optimal.html

[10] **Nešetřil, J.**: A Few Remarks On The History Of MST-Problem, článek je přístupný na adrese: http://citeseer.nj.nec.com/nesetril97few.html

Seznam definic

Seznam obrázků

Seznam tabulek

Seznam grafů

Seznam řešených příkladů