UNIVERZITA PALACKÉHO V OLOMOUCI
PŘÍRODOVĚDECKÁ FAKULTA

# DIPLOMOVÁ PRÁCE

## Metody redukující zaplňování řídkých matic.

**Katedra matematické analýzy a aplikací matematiky**
Vedoucí diplomové práce: **RNDr. Honymír Netuka, Ph.D.**
Vypracoval(a): **Tomáš Mazurka**
Studijní program: N1101 Matematika a její aplikace
Studijní obor Matematika a její aplikace
Forma studia: prezenční
Rok odevzdání: 2018

# BIBLIOGRAFICKÁ IDENTIFIKACE

**Autor:** Tomáš Mazurka

**Název práce:** Metody redukující zaplňování řídkých matic

**Typ práce:** Diplomová práce

**Pracoviště:** Katedra matematické analýzy a aplikací matematiky

**Vedoucí práce:** RNDr. Honymír Netuka, Ph.D.

**Rok obhajoby práce:** 2018

**Abstrakt:** Diplomová práce se zaměří na studium a zpracování algoritmů, které zabraňují většímu zaplňování řídkých matic, přičemž se omezí hlavně na symetrické pozitivně definitní matice. Jádrem práce bude analýza jednotlivých metod a jejich programová realizace zakončená řešením testovacích úloh.

**Klíčová slova:** řídké matice, zaplňování, Algoritmus minimálního stupně, Python

**Počet stran:** 55

**Počet příloh:** 1

**Jazyk:** český

## BIBLIOGRAPHICAL IDENTIFICATION

**Author:** Tomáš Mazurka

**Title:** Fill-in reduction methods for large sparse matrices.

**Type of thesis:** Master's

**Department:** Department of Mathematical Analysis and Application of Mathematics

**Supervisor:** RNDr. Honymír Netuka, Ph.D.

**The year of presentation:** 2018

**Abstract:** This thesis deals with algorithms that reduce fill-in of sparse matrices, focused mostly on sparse symetric positive definite matrices. The main algorithm presented here is Minimum degree algorithm. We introduce some implementations of it, and all these variants will be tested on datas.

**Key words:** sparse matrices, fill-in, Minimum degree algorithm, Python

**Number of pages:** 55

**Number of appendices:** 1

**Language:** Czech

**Prohlášení**

Prohlašuji, že jsem diplomovou práci zpracoval samostatně pod vedením pana RNDr. Honymýra Netuky, Ph.D. a všechny použité zdroje jsem uvedl v seznamu literatury.

V Olomouci dne .......................... ....................................................

podpis

# Obsah

**Úvod** **7**

**1 Základní definice a pojmy** **9**

**2 Základy řešení soustav rovnic** **12**
2.1 Gaussova eliminační metoda . . . . . . . . . . . . . . . . . . . . . . . 13
2.2 LU rozklad . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
2.3 Choleského rozklad . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16

**3 Vlastnosti charakteristické pro řídké matice** **17**
3.1 Struktura a datové schéma řídkých matic . . . . . . . . . . . . . . . 17
3.2 Zaplňování . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
3.3 Zaplňování z pohledu grafů . . . . . . . . . . . . . . . . . . . . . . . 22
3.4 Dosažitelné množiny . . . . . . . . . . . . . . . . . . . . . . . . . . . 26

**4 Algoritmus minimálního stupně** **30**
4.1 Základní algoritmus . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
4.2 MDA pomocí dosažitelných množin . . . . . . . . . . . . . . . . . . 34
4.3 Moje implementace MDA . . . . . . . . . . . . . . . . . . . . . . . . 36

**5 Příklady a testování** **39**
5.1 Příklad na řešení soustavy rovnic . . . . . . . . . . . . . . . . . . . . 40
5.2 testování . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42

**6 Algoritmy** **50**

**Závěr** **54**

**Literatura** **55**

**Poděkování**

Rád bych poděkoval vedoucímu diplomové práce za spolupráci i za čas, který mi věnoval při konzultacích.

# Úvod

Cílem této práce je zkoumání řešení soustav s řídkými maticemi. Budeme pracovat zejména s řídkými, symetrickými, pozitivně definitními velkými maticemi. Zaměříme se hlavně na metodu minimálního stupně, jelikož je to jedna z nejpoužívanějších metod pro práci s velkými řídkými maticemi. Tuto metodu prostudujeme, včetně jejích variant. Tyto varianty naprogramujeme a budeme testovat na námi náhodně vygenerovaných datech. Jako programovací jazyk pro svoji práci jsem zvolil Python. Narozdíl například od Matlabu, v Pythonu se indexuje od nuly, nikoliv od jedničky, a proto i my budeme toto indexování používat v této práci. Z toho plyne, že texty budou v souladu s programovými kódy a indexy budou začínat nulou.

Než se pustíme do samotné metody minimálního stupně, bude třeba si ujasnit několik pojmů a definic. Také si uvedeme několik pojmů z teorie grafů, neboť grafy jsou výhodné pro reprezentaci řídkých matic. S tímto se seznámíme v první kapitole.

V dalších dvou kapitolách se zaměříme na principy, se kterými budeme pracovat. Prvně si vysvětlíme Gaussovu eliminační metodu a rozklady matic, neboť je dobré připomenout základy řešení soustav lineárních rovnic. Nebudeme se sice této problematice věnovat příliš do hloubky, ale alespoň ji nastíníme pro případné čtenáře, kteří s touto problematikou nejsou příliš obeznámeni. Dále se budeme zabývat pamětovými schématy, tedy tím, jak řídké matice uchovávat v paměti efektivnějším způsobem, než jen jako dvourozměrné pole. V poslední části těchto kapitol budeme rozebírat zaplňování jakožto negativní jev, kterému se bude potřeba vyhnout. Právě zde využijeme grafy a jejich vztah s maticemi.

Na otázku, jak řešit zaplňování, nám dá odpověď Algoritmus minimálního stupně, který bude obsahem kapitoly čtvrté. Zde se podíváme na základní implementaci Algoritmu minimálního stupně a pak její upravenou variantu. V závěru této kapitoly bych rád představil svoji vlastní implementaci tohoto algoritmu.

V předposlední kapitole budeme testovat a srovnávat časovou efektivitu jednotlivých algoritmů na řešených příkladech. Jedná se především o obrázky a tabulky, přičemž se přesvědčíme, o kolik je výhodnější pro řídké matice použití Algoritmu minimálního stupně oproti běžnému výpočtu bez použití jakýchkoliv sofistikovanějších metod.

Poslední kapitola obsahuje okomentovaný seznam algoritmů, které jsem v Pythonu vytvořil. Patří sem naprogramované různé verze Algoritmů minimálního stupně, skripty pro testování a také některé pomocné funkce.

# Kapitola 1

# Základní definice a pojmy

V samotném začátku práce se čtenář seznámí s některými důležitými pojmy, které se v této práci budou vyskytovat. Konkrétně se zde jedná o vybrané pojmy z lineární algebry týkající se řídkých matic a taktéž si uvedeme základy teorie grafů a struktur.

Jak již bylo řečeno v úvodu, budeme se zaobírat symetrickými, pozitivně definitními, řídkými maticemi. Pojďme si tedy jednotlivé pojmy rozebrat a objasnit.

**Definice 1.** *Matice $\boldsymbol{A}$ je symetrická, pokud platí, že $\boldsymbol{A}=\boldsymbol{A}^T$.*

**Definice 2.** *Matice $\boldsymbol{A}$ je pozitivně definitní, pokud pro ni platí: $\boldsymbol{x}^T\boldsymbol{A}\boldsymbol{x} > 0$, kde $\boldsymbol{x}$ je libovolný, nenulový, sloupcový vektor.*

Doplnění informací o pozitivně definitních maticích lze nalézt v [5]. Tomuto tématu se již dále věnovat nebudeme, neboť to není obsahem práce.

Posledním termínem, který nám zbývá nadefinovat, je řídká matice. V odborné literatuře lze najít mnoho definic řídkých matic, většina má ovšem společný základ. Tedy, že matice je řídká, obsahuje-li mnoho nul. Kolik přesně nul, se liší. Je možno najít 5, 10 či třeba 20 a nebo klidně i 40%. My bychom si to mohli upřesnit v definici a nadále s tím takto pracovat. Definovat řídké matice tímto způsobem s sebou ovšem nese jisté problémy. Například, pokud si zvolíme hranici 20% a tuto hranici přesáhneme o jedno procento. Budeme mít tedy matici s 21% nenulových prvků. Vyřadíme ji z řídkých matic? Atd.

Proto zvolíme ještě trochu jiný postup. Každá řídká matice může být zpracována jako hustá, a naopak každá hustá matice může být zpracována i algoritmem pro řídké matice. Rozdíl ale bude ve výpočetní náročnosti těchto algoritmů. Tvrzení, že matice je řídká, bude ekvivalentní s tvrzením, že existuje takový algoritmus, který jednoznačne využije vysokého počtu nul v matici aby výpočetní náročnost byla menší, než kdybychom s maticí počítali jako s hustou. [2, str. 1] Zapišme tedy tento poznatek do definice.

**Definice 3.** *Matice $\boldsymbol{A}$ je řídká, pokud existuje algoritmus, jehož výpočetní náročnost je menší, než kdybychom použili stejnou variantu algoritmu pro hustou matici.*

Jako příklad si můžeme vzít algoritmus, který nebude dělat nic jiného, než že po prvcích sečte dvě pouze diagonální matice $\mathbf{A}$ a $\mathbf{B}$, přičemž obě budou čtvercové, řádu 100. Pokud budeme matice uvažovat jako husté, budeme sčítat $100 \times 100$, tedy 10000 prvků a bude tak potřeba řádově i stejné množství operací. Ale vzhledem k tomu, že nenulové prvky je potřeba sečíst jen na diagonále, která má pouze 100 prvků, tak jistě nebude problém napsat algoritmus, který sečte pouze diagonály těchto matic a zbytek nechá roven nule. Bude tak potřeba řádově jen 100 operací. Tj. $100\times$ méně. Vidíme, že tedy existuje algoritmus, který využil nul v matici a matice A a B tak můžeme považovat za řídké.

Nyní si uvedeme několik pojmů, které budou souviset s grafy. Grafy budeme hojně využívat, protože jsou pro popis řídkých matic velmi vhodné a přínosné.

**Definice 4.** *Graf $\boldsymbol{G} = (V, E)$ je množina uzlů $V = \{1, ..., n\}$ a hran $E = \{\{i, j\} : I, J \in V\}$ spojujících tyto uzly.*

Dále řekneme, že graf je *neorientovaný*, pokud hrany $\{i, j\}$ a $\{j, i\}$ jsou totožné. Jedná-li se o dvě různé hrany, řekneme, že graf je *orientovaný*. Může se tak stát, že v orientovaném grafu hrana $\{i, j\}$ existuje a hrana $\{j, i\}$ nikoli.

**Definice 5.** *Matice sousednosti $\boldsymbol{A}$ grafu $\boldsymbol{G}$ je binární $n \times n$ matice, ve které je $a_{ij} = 1$, pokud hrana $\{i, j\} \in E$, a $a_{ij} = 0$ v opačném případě.*

Tedy platí, že pro neorientovaný graf je matice sousednosti symetrická.[1, str. 4]

My ovšem nebudeme s matící sousednosti jako celkem příliš pracovat. Více využijeme jednotlivé seznamy sousedů pro jednotlivé uzly. Sousedy uzlu $x$ budeme značit $Adj(x)$ a tyto sousedy zapíšeme do seznamu. V postatě se jedná o to samé, liší se jen způsob reprezentace. Seznamy sousedů mnohem lépe korespondují s řídkými schématy matic. Pokud tedy budeme mít matici sousednosti $\mathbf{A}$, kde

$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}.$$

Pak s ní budeme pracovat jako se seznamem o 4 seznamech, kde první seznam bude obsahovat seznam čísel uzlů sousedů prvního uzlu. Dostaneme

$$seznam_sousedu = [[1], [0, 3], [3], [1, 2]].$$

Tedy uzel $x_0$ má za sousedy pouze uzel $x_1$, uzel $x_1$ má za sousedy dva uzly a to $x_0, x_3$ a tak dále.

**Definice 6.** *Stupeň uzlu je počet sousedů daného uzlu.*

**Definice 7.** *Cesta z $x$ do $y$ délky $l > 0$ je uspořádaná množina, navzájem různých, $l + 1$ uzlů $(v_1, v_2, ..., v_{l+1})$ tak, že $v_{l+1} \in Adj(v_i)$ Pro $i = 1, 2, ..., l$ a kde $v_1 = x$ a $v_{l+1} = y$*

# Kapitola 2

# Základy řešení soustav rovnic

V této krátké kapitole se zběžně seznámíme se základy řešení soustav lineárních rovnic (SLR), aby i méně zkušený čtenář byl schopen zvládnout pojmy z kapitol následujících. Nebudeme zde rozebírat řešení rovnic dopodrobna, neboť to není cílem této diplomové práce.

**Definice 8.** *Soustavou $n$ lineárních rovnic o $n$ neznámých $x_1, x_2, ..., x_n$ rozumíme:*

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + ... + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + ... + a_{2n}x_n &= b_2 \\ &... \\ a_{n1}x_1 + a_{n2}x_2 + ... + a_{nn}x_n &= b_n, \end{aligned} \tag{2.1}$$

*kde $a_{11}, ..., a_{nn}, b_1, ..., b_n$ jsou daná reálná císla.*

Soustavu lze zapsat i maticově ve tvaru $\mathbf{Ax} = \mathbf{b}$, kde $\mathbf{A} = (a_{ij})_{i,j=1}^n, \mathbf{x} = [x_1, ..., x_n]^T$ a $\mathbf{b} = [b_1, ..., b_n]^T$.

Základní myšlenkou přímých metod pro řešení soustav rovnic, je naši soustavu (2.1) převest do tvaru:

$$\begin{aligned} u_{11}x_1 + u_{12}x_2 + ... + u_{1n}x_n &= c_1 \\ u_{22}x_2 + ... + u_{2n}x_n &= c_2 \\ &... \\ u_{nn}x_n &= c_n. \end{aligned} \tag{2.2}$$

Soustavu samozřejmě lze zapsat i maticově ve tvaru $\mathbf{U}\mathbf{x} = \mathbf{c}$, kde $\mathbf{U} = (u_{ij})_{i,j=1}^{n}$ je horní trojúhelníková matice, $\mathbf{x} = [x_1, ..., x_n]^T$ a $\mathbf{c} = [c_1, ..., c_n]^T$.

## 2.1. Gaussova eliminační metoda

Gaussova eliminační metoda, zkracujeme GEM, je základní přímou metodou řešení SLR. Má dvě fáze. Její první fáze nám ukazuje, jak převést soustavu (2.1) do (2.2). V druhé fázi řešíme soustavu (2.2) od poslední rovnice směrem k první a dostaneme tak požadovaný vektor $\mathbf{x}$ jakožto řešení naší soustavy (2.1).

Nyní si tedy popíšeme GEM. Budeme popisovat jen základní metodu bez výběru pivota. Pivotování je proces, kdy prohazujeme v matici řádky a sloupce ekvivalentními úpravami tak, aby matice byla pozitivně definitní a numericky stabilní. Nicméně zde pracujeme již z pozitivně definitními maticemi, netřeba tedy pivotovat za tímto účelem. Platí tak, že v naší soustavě (2.2) je každý koeficient $a_{ii}$ nenulový. Je-li $\mathbf{x} = [x_1, ..., x_n]^T$ řešením soustavy (2.1), pak je také řešením soustavy:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + ... + a_{1n}x_n &= b_1 \\ a_{22}^{(1)}x_2 + ... + a_{2n}^{(1)}x_n &= b_2^{(1)} \\ &... \\ a_{n2}^{(1)}x_2 + ... + a_{nn}^{(1)}x_n &= b_n^{(1)}. \end{aligned} \tag{2.3}$$

Děje se tak z důvodu, že od soustavy (2.1) jsme k soustavě (2.3) došli řádkově ekvivalentními úpravami. Konkrétně druhou rovnici této soustavy (2.3) jsme dostali tak, že jsme odečetli od druhé rovnice v (2.1) $(a_{21}/a_{11})$–násobek rovnice první. Třetí rovnici jsme dostali tak, že jsme odečetli $(a_{31}/a_{11})$–násobek první rovnice od rovnice třetí atd. Obecně tedy

$$a_{ik}^{(1)} = a_{ik} - a_{i1}a_{11}^{-1}a_{1k}, \qquad k = 2, ..., n,$$

a na pravé straně

$$b_i^{(1)} = b_i - a_{i1}a_{11}^{-1}b_1, \qquad k = 2, ..., n.$$

Tímto způsobem jsme vyeliminovali všechny prvky pod diagonálou v prvním sloupci a přesuneme se k dalšímu sloupci. A takto pokračujeme dál, až dostaneme soustavu tvaru (2.2), což je soustava s horní trojúhelníkovou maticí, která má stejné řešení jako původní soustava (2.1).

## 2.2. LU rozklad

Cílem LU rozkladu je najít takové regulární, po řadě dolní a horní trojúhelníkové matice **L** a **U**, aby platilo **A=LU**. Uvedeme si, jak naši SLR (2.1) vyřešit pomocí LU rozkladu. V řešení SLR se LU rozklad uplatní následovně. Za **A** dosadíme **LU** do původní soustavy rovnic **Ax=b**, dostaneme **LUx=b**. Označíme **Ux=c** a dostaneme soustavu rovnic **Lc=b**. Soustavu rovnic **Lc=b** vyřešíme snadno, vektor **b** známe a stejne tak známe i dolní trojúhelníkovou matici **L**. Začneme první rovnicí, ze které vypočítáme $c_1$. V druhé rovnici máme neznámé $c_1$ a $c_2$, tedy $c_2$ vypočítáme, protože $c_1$ jsme již vypočítali v minulém kroku a tak za něj pouze dosadíme. Takto postupujeme až k poslední rovnici, čímž dostaneme celý vektor **c**.

Nyní můžeme začít řešit soustavu **Ux=c** (znázorněna v (2.2)). Řešení soustavy **Ux=c** se nazývá zpětný chod a jejím výsledkem je hledaný vektor **x**, který je řešením původní soustavy rovnic **Ax=b**. Postup řešení soustavy **Ux=c** je obdobný jako postup řešení soustavy **Lc=b** s tím rozdílem, že zde je matice **U** horní trojúhelníková. Proto nezačínáme první rovnicí, ale poslední. Od toho název zpětný chod. Z poslední rovnice vypočítáme $x_n$. V předposlední rovnici máme neznámé $x_n$ a $x_{n-1}$, tedy $x_{n-1}$ vypočítáme, protože $x_n$ jsme již vypočítali v minulém kroku a tak za něj pouze dosadíme. Takto postupujeme až k první rovnici a dostaneme celý vektor **x**.

Jak budou vypadat prvky matic **L** a **U**? Podívejme se na výpočet prvků matic $\mathbf{L} = (l_{ij})$ po sloupcích a $\mathbf{U} = u_{ij}$ po řádcích.

Položíme

$$u_{11} = a_{11}.$$

Pro $i = 1, ..., n$:

$$l_{i1} = \frac{a_{i1}}{u_{11}}.$$

Pro $r = 2, ..., n$:

$$u_{1r} = a_{1r}.$$

Pro $i = 2, ..., r$:

$$u_{ir} = a_{ir} - \sum_{j=1}^{i-1} l_{ij} u_{jr}.$$

Pro $i = r + 1, ..., n$:

$$l_{ir} = \frac{1}{u_{rr}}(a_{ir} - \sum_{j=1}^{r-1} l_{ij} u_{jr}).$$

Pro $k = 1, ..., n$:

$$l_{kk} = 1.$$

Zbylé prvky jsou pak rovny nule. Algoritmus, který jsme si právě uvedli, je odvozen z maticového násobení. Vyjdeme z původní rovnice **A=LU**, kde **A** je matice řádu $n$, kterou chceme rozložit. **L** je pro nás zatím neznámá matice, o které ale víme, že je dolní trojúhelníková. O matici **U** víme jen to, že je horní trojúhelníková. Pro maticové násobení platí $a_{ij} = \sum_{k=1}^{n} l_{ik} u_{kj}$. Prvky matice **A** známe. Z těchto informací jsme schopni postupně dopočítat prvky matic **L** a **U**.

Když budeme řešit soustavu linárních rovnic pomocí LU rozkladu, tak řádově provedeme stejně operací jako při použití Gaussovy eliminační metody. Z hlediska složitosti jsou tedy tyto metody ekvivalentní. Použití LU rozkladu je ovšem výhodnejší v případě, kdy řešíme více soustav lineárních rovnic se stejnou maticí a mění se nám pouze vektor pravé strany. Pak se totiž nejnáročnejší část výpočtu, tedy výpočet matic **L** a **U**, provede pouze jednou. [6, str. 94].

## 2.3. Choleského rozklad

Choleského rozklad je varianta LU rozkladu pro symetrické matice. Hledáme takovou matici $\mathbf{L}$, pro kterou platí $\mathbf{L}{*}\mathbf{L}^T = \mathbf{A}$. Tedy $\mathbf{L}$ je dolní trojúhelníková matice stejně jako u LU rozkladu, ale $\mathbf{U}$ je nyní rovno $\mathbf{L}^T$. Vzhledem k tomu, že zde budeme pracovat pouze se symetrickými maticemi, tak právě Choleského rozklad je metoda, jíž budeme při řešení rovnic využívat. Za $\mathbf{A}$ tedy nyní dosadíme $\mathbf{L}{*}\mathbf{L}^T$ a dostaneme tak $\mathbf{L}\mathbf{L}^T\mathbf{x} = \mathbf{b}$. A stejně jako u LU rozkladu si zde označíme $\mathbf{L}^T\mathbf{x} = \mathbf{c}$ a dostaneme tak $\mathbf{L}\mathbf{c} = \mathbf{b}$. Tuto soustavu vyřešíme a po získání vektoru $\mathbf{c}$ vyřešíme zpětným chodem soustavu $\mathbf{L}^T\mathbf{x} = \mathbf{c}$.

Matici $\mathbf{L}$ opět získáme pomocí maticového násobení v následujícím dvojitém $for$ cyklu.

Pro $j = 0, ..., n$:

$$l_{jj} = \sqrt{a_{jj} - \sum_{i=0}^{j-1} l_{ji}^2}$$

Pro $i = j+1, ..., n$:

$$l_{ij} = \frac{a_{ij} - \sum_{t=0}^{j} l_{it} l_{jt}}{l_{jj}}$$

Zbylé prvky jsou zase rovny nule. Stejně jako LU rozklad je i Choleského rozklad odvozen z maticového násobení.

# Kapitola 3

# Vlastnosti charakteristické pro řídké matice

## 3.1. Struktura a datové schéma řídkých matic

V této sekci se budeme věnovat různým způsobům, jak mít řídkou matici uloženou v paměti počítače. Vezměme si tedy matici $\mathbf{A}$, na níž v této sekci budeme ukazovat jednotlivé způsoby uložení v paměti počítače. Vzhledem k tomu, že každá řídká matice je vlastně hustá matice se spoustou nul, můžeme každou řídkou matici v počítači uložit úplně stejně, jako matici hustou.

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 0 & 0 & 3 & 0 & 0 & 4 \\ 0 & 5 & 0 & 6 & 0 & 0 & 0 & 0 \\ 0 & 0 & 7 & 0 & 0 & 8 & 0 & 0 \\ 0 & 9 & 0 & 10 & 0 & 0 & 11 & 0 \\ 12 & 0 & 0 & 0 & 13 & 0 & 0 & 0 \\ 0 & 14 & 0 & 0 & 0 & 15 & 0 & 0 \\ 0 & 0 & 0 & 16 & 0 & 0 & 17 & 0 \\ 0 & 0 & 18 & 0 & 0 & 0 & 19 & 20 \end{bmatrix}.$$

Je pochopitelné, že tento způsob nepřinese žádné inovace, neboť jsme nijak nevyužili množství nul v matici a tím v podstatě ani nesplnili definici řídké matice.
Jedním z možných způsobu uložení řídké matice je takzvaná triplet form neboli trojice seznamů. Ta může vypadat následovně:

$x = \{0, 0, 0, 2, 3, 3, 3, 4, 0, 1, 1, 2, 6, 5, 7, 7, 4, 7, 6, 5\}$,
$y = \{0, 1, 4, 5, 1, 3, 6, 0, 7, 1, 3, 2, 6, 1, 7, 6, 4, 2, 4, 5\}$,
$hodnoty = \{1, 2, 3, 8, 9, 10, 11, 12, 4, 5, 6, 7, 17, 14, 20, 19, 13, 18, 16, 15\}$.

Tato forma je velice jednoduchá na vytvoření, ale ještě stále zabírá v paměti zbytečně více místa, než je potřeba, a v algoritmech se příliš nepoužívá. [1, str. 8] V podstatě je to jen zápis hodnot z matice do vektoru a k nim jejich řádkový a sloupcový index, přičemž vynecháme prvky matice rovné nule. V tomto seznamovém zápise tedy vždy hodnotě $hodnota(k)$ odpovídá $x(k)$ jako řádkový index a $y(k)$ jako sloupcový index. Je tedy možno hodnoty zapisovat v jakémkoliv uspořádání.

Pokud ovšem budeme tuto trojici vektorů sestavovat tak, že matici budeme projíždět po sloupcích, dostaneme následující výsledek:

$x = \{0, 4, 0, 1, 3, 5, 2, 7, 1, 3, 6, 0, 4, 2, 5, 3, 6, 7, 0, 7\}$,
$y = \{0, 0, 1, 1, 1, 1, 2, 2, 3, 3, 3, 4, 4, 5, 5, 6, 6, 6, 7, 7\}$,
$hodnoty = \{1, 12, 2, 5, 9, 14, 7, 18, 6, 10, 16, 3, 13, 8, 15, 11, 17, 19, 4, 20\}$.

Když se podíváme na $y$, vidíme, jak jsou tyto indexy postupně uspořádány. To je dáno tím, že jsme matici projížděli po sloupcích. (Kdybychom ji projíždeli po řádcích, tak se totéž stane u $x$.)

Právě tohoto poznatku lze využít a dostat tak již hodně používanou CCS formu (Compressed column storage). Někdy také jako CSC (Compressed sparse column storage). A jak již název napovídá, jedná se o komprimovanou, hojně používanou datovou strukturu k zápisu řídkých matic, získanou pomocí průchodu matice po sloupcích. Existuje také CRS (Compressed row storage), jenž je získána pomocí průchodu po řádcích. V případě naší matice **A** vypadá CCS následovně:

$x = \{0, 4, 0, 1, 3, 5, 2, 7, 1, 3, 6, 0, 4, 2, 5, 3, 6, 7, 0, 7\}$,
$y = \{0, 2, 6, 8, 11, 13, 15, 18, 20\}$,

$hodnoty = \{1, 12, 2, 5, 9, 14, 7, 18, 6, 10, 16, 3, 13, 8, 15, 11, 17, 19, 4, 20\}$.

Lze vidět, že $x$ a *hodnoty* zůstaly stejné. Avšak nyní je již $y$ mnohem kratší a taktéž se změnil jeho význam. Pokud tedy máme řídkou matici o rozměrech $m \times n$, která obsahuje $c$ nenulových čísel, potom jako CCS ji reprezentujeme pomocí:

—vektoru *hodnot*, obsahujícího právě $c$ nenulových hodnot z matice, většinou formátu double.

—vektoru $x$, který obsahuje taktéž $c$ prvků, v tomto případě to jsou však indexy, jedná se tudíž o celočíselné hodnoty.

—vektoru $y$, délky $n + 1$. Tento vektor nám říká, od kolikátého indexu pole $x$ začíná další sloupec naší matice $\mathbf{A}$. První prvek je vždy 0 a poslední je vždy roven počtu prvků matice.

Platí tedy, že například $hodnota(4) = 9$ je v matici na řádku $x(4) = 3$, což je 4. řádek, protože číslujeme od nuly. A v 1. sloupci neboť číslo 4 (index pole *hodnota*) leží ve vektoru $y$ mezi čísly 2 a 6. Vezme se to menší z nich, což je číslo 2 a jeho index je 1.

## 3.2. Zaplňování

V angličtině je tento fenomém známý pod pojmem Fill-in. A jedná o stav, kdy nám v matici přibývají nenulové prvky. Zaplňování nastává ve fázi, kdy řešíme soustavu rovnic, reprezentovanou maticí, pomocí Gaussovy eliminační metody. Neboli, když rozkládáme matici pomocí LU rozkladu, v případě symetrické matice je to Choleského rozklad. Připomeňme si, že v $k$-tém kroku provádíme následující:

$$l_{i,j} = \frac{a_{i,j} - \sum_{t=0}^{n} l_{it} l_{jt}}{l_{j,j}}$$

A tedy i pokud v původní matici na pozici $a_{i,j}$ byla nula, po rozkladu na stejné pozici $l_{i,j}$ nula již být nemusí, pokud suma $\sum_{t=0}^{n}(l_{i,t} * l_{j,t}))$ bude nenulová. A právě tyto prvky $l_{i,j}$, přičemž $a_{i,j} = 0$, jsou nazvány zaplněním. Pojďme si uvést

příklad. Vezměme si matici **A**, která vypadá následovně. Jelikož řešíme pouze pozice prvků, netřeba znát přesné hodnoty. Tuto matici budeme rozkládat Choleského rozkladem, a dostaneme tak matici **L**.

$$\mathbf{A} = \begin{bmatrix} x & x & x & x & x & 0 & 0 & 0 \\ x & x & 0 & 0 & 0 & 0 & 0 & x \\ x & 0 & x & 0 & 0 & 0 & 0 & 0 \\ x & 0 & 0 & x & 0 & x & 0 & 0 \\ x & 0 & 0 & 0 & x & x & x & 0 \\ 0 & 0 & 0 & x & x & x & 0 & 0 \\ 0 & 0 & 0 & 0 & x & 0 & x & 0 \\ 0 & x & 0 & 0 & 0 & 0 & 0 & x \end{bmatrix}.$$

$$\mathbf{L} = \begin{bmatrix} x & & & & & & & \\ x & x & & & & & & \\ x & \bullet & x & & & & & \\ x & \bullet & \bullet & x & & & & \\ x & \bullet & \bullet & \bullet & x & & & \\ 0 & 0 & 0 & x & x & x & & \\ 0 & 0 & 0 & 0 & x & \bullet & x & \\ 0 & x & \bullet & \bullet & \bullet & \bullet & \bullet & x \end{bmatrix}.$$

Symbol $\bullet$ nám značí zaplnění způsobené rozkladem matice **A**.
Cílem metod pro řídké matice je však vyhnout se početním operacím s nulovými prvky, kterých chceme mít v matici co nejvíce. Je proto snahou se zaplňování vyhnout. Zaplňování je tudíž negativní jev, a je ovlivněno výběrem pivota.
Problém zaplňování se dá nejlépe demonstrovat na maticích typu šipky. Vezměme si takovou matici **A1**, jež vypadá následovně.

$$\mathbf{A1} = \begin{bmatrix} x & x & x & x & x & x & x & x \\ x & x & 0 & 0 & 0 & 0 & 0 & 0 \\ x & 0 & x & 0 & 0 & 0 & 0 & 0 \\ x & 0 & 0 & x & 0 & 0 & 0 & 0 \\ x & 0 & 0 & 0 & x & 0 & 0 & 0 \\ x & 0 & 0 & 0 & 0 & x & 0 & 0 \\ x & 0 & 0 & 0 & 0 & 0 & x & 0 \\ x & 0 & 0 & 0 & 0 & 0 & 0 & x \end{bmatrix}.$$

Choleského rozklad **L** této matice bude mít největší možné zaplnění.

$$\mathbf{L} = \begin{bmatrix} x & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ x & x & 0 & 0 & 0 & 0 & 0 & 0 \\ x & \bullet & x & 0 & 0 & 0 & 0 & 0 \\ x & \bullet & \bullet & x & 0 & 0 & 0 & 0 \\ x & \bullet & \bullet & \bullet & x & 0 & 0 & 0 \\ x & \bullet & \bullet & \bullet & \bullet & x & 0 & 0 \\ x & \bullet & \bullet & \bullet & \bullet & \bullet & x & 0 \\ x & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & x \end{bmatrix}.$$

Přitom stačí, abychom matici **A1** správnou permutací **P** přerovnali do podoby **A2**.

$$\mathbf{A2} = \begin{bmatrix} x & 0 & 0 & 0 & 0 & 0 & 0 & x \\ 0 & x & 0 & 0 & 0 & 0 & 0 & x \\ 0 & 0 & x & 0 & 0 & 0 & 0 & x \\ 0 & 0 & 0 & x & 0 & 0 & 0 & x \\ 0 & 0 & 0 & 0 & x & 0 & 0 & x \\ 0 & 0 & 0 & 0 & 0 & x & 0 & x \\ 0 & 0 & 0 & 0 & 0 & 0 & x & x \\ x & x & x & x & x & x & x & x \end{bmatrix}.$$

Výsledek Choleského rozkladu **L2** bude mnohem uspokojivější.

$$\mathbf{L2} = \begin{bmatrix} x & 0 & 0 & 0 & 0 & 0 & 0 & x \\ 0 & x & 0 & 0 & 0 & 0 & 0 & x \\ 0 & 0 & x & 0 & 0 & 0 & 0 & x \\ 0 & 0 & 0 & x & 0 & 0 & 0 & x \\ 0 & 0 & 0 & 0 & x & 0 & 0 & x \\ 0 & 0 & 0 & 0 & 0 & x & 0 & x \\ 0 & 0 & 0 & 0 & 0 & 0 & x & x \\ x & x & x & x & x & x & x & x \end{bmatrix}.$$

Zde nedošlo k zaplnění vůbec. To nazýváme perfektní eliminace.

Jak se lze dočíst v [3, str. 99], jde vidět, že zaplňování závisí na vlastnostech matice. V obecném případě, pokud není matice pozitivně definitní, musíme balancovat mezi zaplňováním a numerickou stabilitou. Je-li naše řídká matice symetrická a pozitivně definitní, máme svobodu v pivotování za účelem minimalizace zaplnění. Navíc můžeme už s předstihem určit, která místa se nám zaplní. Což se nazývá symbolická faktorizace. I na našem příkladu v této kapitole jsme

mohli vidět, že konkrétní hodnota jednotlivých prvků nemá vliv na to, která místa se zaplní. Zaplňování ovlivňuje pouze struktura matice a pivotování. Neboli Graf této matice a jeho očíslování.

## 3.3. Zaplňování z pohledu grafů

V předchozí sekci jsme si zaplňování definovali a určili jsme, co jej způsobuje. Nyní se podíváme na to, jak s ním pracovat z pohledu grafů. Bude pro nás důležitá následující věta. Vezměme si posloupnost grafů $\mathbf{G}^{(i)}$ tak, že $\mathbf{G}^{(0)} = \mathbf{G}$ a $\mathbf{G}^{(i)}$ je graf v $i$-tém kroku eliminace.

**Věta 1.** *$\boldsymbol{G}^{(i+1)}$ získáme z $\boldsymbol{G}^{(i)}$ odstraněním uzlu $x_i$ z grafu a odstraněním všech hran vedoucích k tomuto uzlu. Zároveň však do grafu přidáme hrany mezi všemi zbývajícími sousedy daného uzlu tak, aby byl každý s každým propojen.*

Důkaz čtenář najde například v [3, str. 102]. Nicméně hlavní myšlenka je následující. Vezměme si první krok GEM. Pak platí, že

$$a_{i,j}^{(1)} = a_{i,j}^{(0)} - \frac{a_{i,1}^{(0)} * a_{1,j}^{(0)}}{a_{1,1}}.$$

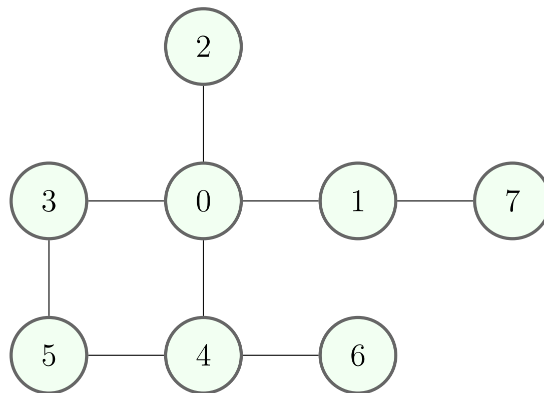
Vidíme, že nové $a_{i,j}$ je nenulové, pokud původní $a_{i,j}$ bylo nenulové a nebo pokud jsou $a_{i,1}^{(0)} \neq 0$ a zároveň $a_{1,j}^{(0)} \neq 0$. A právě druhá varianta nám vlastně říká, že oba uzly $x_i$ a $x_j$ jsou sousedé uzlu $x_1$. Tedy, pokud eliminujeme uzel $x_1$, vytvoří se mezi jeho sousedy nová hrana, která je reprezentována právě pomocí nenulového $a_{i,j}$.

Zaplňení v teorii grafů značíme do eliminačních grafů (elimination graphs), značíme $\mathbf{G}_F$, $\mathbf{F}$=$\mathbf{L}$+$\mathbf{L}^T$. A tedy $\mathbf{G}_F = (V, E^F)$. Demonstrujme na následujícím

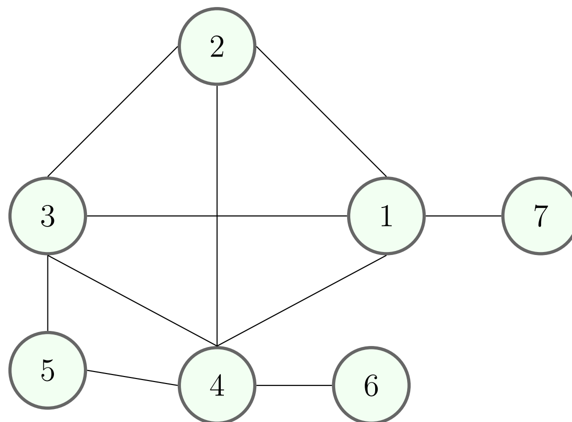
příkladu. Vezměme si matici **A**, se kterou jsme se již setkali v předešlé sekci 3.2.

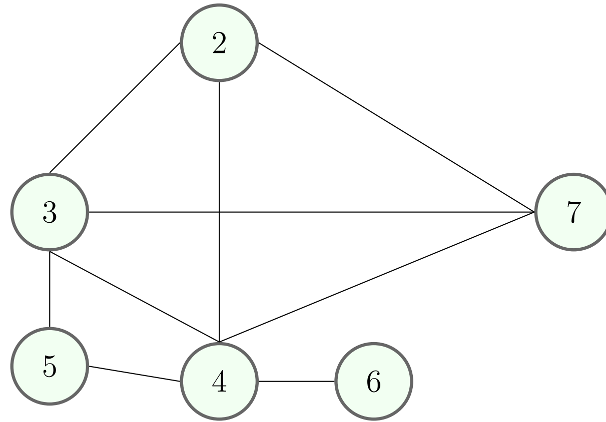
$$\mathbf{A} = \begin{bmatrix} x & x & x & x & x & 0 & 0 & 0 \\ x & x & 0 & 0 & 0 & 0 & 0 & x \\ x & 0 & x & 0 & 0 & 0 & 0 & 0 \\ x & 0 & 0 & x & 0 & x & 0 & 0 \\ x & 0 & 0 & 0 & x & x & x & 0 \\ 0 & 0 & 0 & x & x & x & 0 & 0 \\ 0 & 0 & 0 & 0 & x & 0 & x & 0 \\ 0 & x & 0 & 0 & 0 & 0 & 0 & x \end{bmatrix}.$$

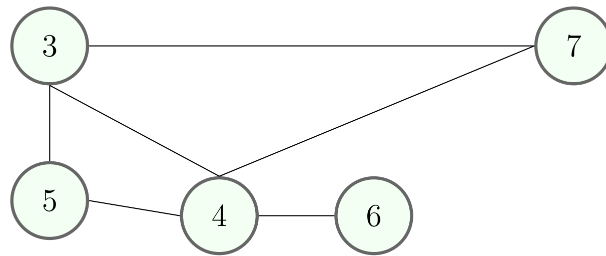
Její graf bude vypadat následovně.

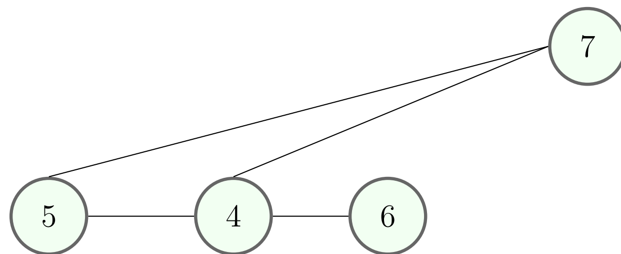


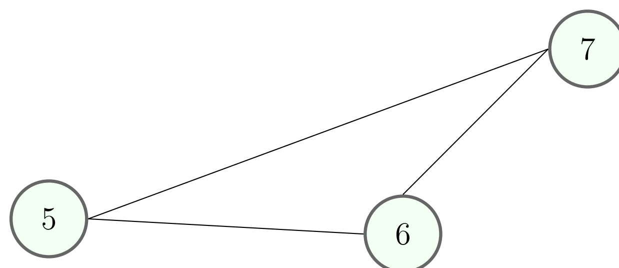

Ukážeme si, jak se graf bude měnit postupnou eliminací uzlů $x_0, x_1, x_2, ..., x_7$.

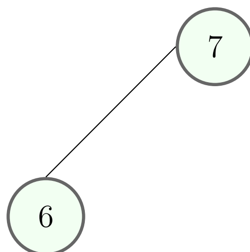



2
3
7
5
4
6

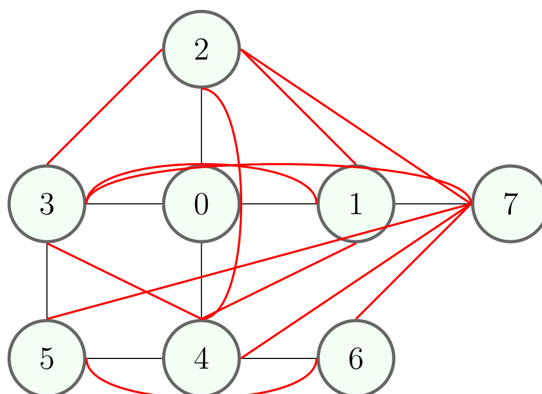

3
7
5
4
6


7
5
4
6




Výsledný eliminační graf $\mathbf{G}_F$ pak bude vypadat následovně.




Jak jsme si v 3.2 ukázali, naší matici $\mathbf{A}$ odpovídá matice $\mathbf{L}$. Pokud tedy uděláme $\mathbf{L}+\mathbf{L}^T$ (jen ji symetricky doplníme), tak tato matice bude odpovídat našemu

eliminačnímu grafu $\mathbf{G}_F$. Nebude-li uvedeno jinak, matici $\mathbf{L}+\mathbf{L}^T$ budeme značit $\mathbf{F}$. Pro naši matici $\mathbf{A}$ tak dostáváme $\mathbf{F(A)} = (\mathbf{L}+\mathbf{L}^T)\mathbf{(A)}$, přičemž všechny nové hrany (červené) odpovídají v matici $\mathbf{F(A)}$ symbolům $\bullet$.

$$\mathbf{F(A)} = \begin{bmatrix} x & x & x & x & x & 0 & 0 & 0 \\ x & x & \bullet & \bullet & \bullet & 0 & 0 & x \\ x & \bullet & x & \bullet & \bullet & 0 & 0 & \bullet \\ x & \bullet & \bullet & x & \bullet & x & 0 & \bullet \\ x & \bullet & \bullet & \bullet & x & x & x & \bullet \\ 0 & 0 & 0 & x & x & x & \bullet & \bullet \\ 0 & 0 & 0 & 0 & x & \bullet & x & \bullet \\ 0 & x & \bullet & \bullet & \bullet & \bullet & \bullet & x \end{bmatrix}.$$

Pro přehlednost si následující poznatek zapíšeme do definice.

**Definice 9.** *$\mathbf{F(A)} = (\mathbf{L}+\mathbf{L}^T)\mathbf{(A)}$ nazveme maticí zaplnění matice $\mathbf{A}$. Ta zároveň odpovídá eliminačnímu grafu $\mathbf{G}_F$.*

V anglické literatuře se $\mathbf{F(A)}$ často nazývá filled matrix a nebo factorized matrix.

## 3.4. Dosažitelné množiny

V předešlé sekci jsme mohli pozorovat, jak získat $\mathbf{G}_F$ za pomocí posloupností $\mathbf{G}_i = (V_i, E_i)$. Ovšem nejen v teorii, ale i při praktických výpočtech, by bylo dobré míti k dispozici postup, jak $\mathbf{G}_F$ získat přímo z originálního grafu $\mathbf{G}_0 = (V_0, E_0)$, bez nutnosti postupovat rekurzivně po jednotlivých krocích. A právě to je cílem této sekce.

Zde si ukážeme, jak získat eliminační graf $\mathbf{G}_F$ přímo z originálního grafu $\mathbf{G}_0 = (V_0, E_0)$, aniž bychom museli tvořit posloupnosti grafů. Jinak řečeno nás zajímá, jak přímo z původního grafu určit, které nové hrany vzniknou, a tedy, která místa v matici se zaplní. Tento poznatek posléze využijeme i při zjišťování sousedů uzlů v $i$-tém kroku eliminace.

Budeme zde využívat stejný graf jako v minulé sekci. Začneme tím, že podrobněji prostudujeme, jak vznikla hrana $\{4, 7\}$, jedná se o hranu mezi uzly $x_4$

a $x_7$. Tato hrana vznikla v grafu poté, co jsme eliminovali uzel $x_1$. Tím se nám z hran $\{1,4\}$ a $\{1,7\}$ stala hrana $\{4,7\}$. Ale ani hrana $\{1,4\}$ v původním grafu nebyla obsažena. Ta vznikla při eliminaci uzlu $x_0$, kdy se z $\{0,4\}$ a $\{0,1\}$ vytvořila hrana $\{1,4\}$. Vidíme tak, že za vnikem hrany $\{4,7\}$ stojí ve skutečnosti cesta $(x_4, x_0, x_1, x_7)$. Právě tohle je motivací k použití dosažitelných množin, které si nyní představíme.

**Definice 10.** *Nechť $S$ je podmnožina množiny uzlů $V$. Nechť $x \notin S$. Potom řekneme, že $x$ je dosažitelné z uzlu $y$ přes množinu $S$, pokud existuje taková cesta $(y, u_1, u_2, ..., u_k, x)$ z $x$ do $y$, pro kterou $u_i \in S$ pro $1 < i < k$ .*

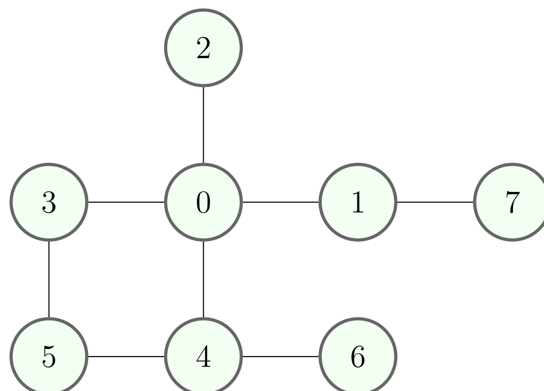
Poznamenejme, že může nastat i $k = 0$. Tedy každý soused uzlu $y$, který není obsažen v množině $S$, je dosažitelný z $y$ přes $S$.
Dosažitelnou množinu z uzlu $y$ přes množinu $S$ budeme tedy značit:

$$Reach(y, S) = \{x \notin S |\ x \text{ je dosažitelné z } y \text{ přes } S\}.$$

Nyní, když máme k dispozici funkci $Reach(y, S)$, můžeme nově vzniklé hrany v eliminačním grafu $\mathbf{G}_F$ určit snadno z původního grafu $\mathbf{G}_0 = (V_0, E_0)$, bez nutnosti sestavování celé posloupnosti grafů, jako jsme to dělali v sekci 3.3. Shrňme tento poznatek do následující věty, jejíž důkaz lze najít v [4, str. 113].

**Věta 2.** $\boldsymbol{E^F} = \{\{x_i, x_j\} |\ x_j \in Reach(x_i, \{x_0, x_1, ..., x_{i-1}\})\}$.

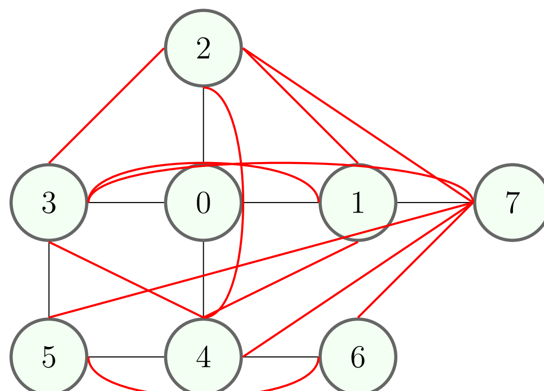
Věta nám říká, že pro eliminační graf dáváme do množiny $S$ všechny již zpracované a očíslované uzly. Jelikož číslujeme od začátku, tak $S$ obsahuje vždy všechny uzly s číslem menším, než je číslo uzlu, pro který aktuálně zjisťujeme jeho dosah. Pojďme si větu demonstrovat na příkladu. Vezměme si náš graf matice $\mathbf{A}$.




Jak bude vypadat *Reach* pro jednotlivé uzly? Jak již bylo zmíněno, množina $S$ vypadá následovně: $S_i = \{x_0, x_1, ..., x_{i-1}\}$.

$$
\begin{aligned}
&Reach(x_0, S_0) = \{x_1, x_2, x_3, x_4\}, && S_0 = \{\} \\
&Reach(x_1, S_1) = \{x_2, x_3, x_4, x_7\}, && S_1 = \{x_0\} \\
&Reach(x_2, S_2) = \{x_3, x_4, x_7\}, && S_2 = \{x_0, x_1\} \\
&Reach(x_3, S_3) = \{x_4, x_5, x_7\}, && S_3 = \{x_0, x_1, x_2\} \\
&Reach(x_4, S_4) = \{x_5, x_6, x_7\}, && S_4 = \{x_0, ..., x_3\} \\
&Reach(x_5, S_5) = \{x_6, x_7\}, && S_5 = \{x_0, ..., x_4\} \\
&Reach(x_6, S_6) = \{x_7\}, && S_6 = \{x_0, ..., x_5\} \\
&Reach(x_7, S_7) = \{\}, && S_7 = \{x_0, ..., x_6\}.
\end{aligned}
$$

Připomeňme si, jak vypadal náš původní graf po eliminaci.




Můžeme zde vidět, že výše uvedené schéma funkce *Reach* splňuje náledující. $Reach(x_i, S_i)$ nám vždy dá sousedy uzlu $x_i$ v $i$-tém Gaussovy.

Navíc, množina $\{Reach(x_i, S_i), S_i\}$ nám dá všechny sousedy uzlu $x_i$ ve výsledném eliminačním grafu $\mathbf{G}_F$.

Pozorujeme, že se nám podařilo strukturu $\mathbf{L}$ zcela popsat pouze výchozím grafem $\mathbf{G}_0$ a funkcí *Reach*. Shrňme si tyto poznatky do věty.

**Věta 3.** *Nechť y je uzel v eliminačním grafu $\boldsymbol{G}_i = (V_i, E_i)$. Množina sousedů uzlu y v grafu $\boldsymbol{G}_i$ je dána pomocí $Reach(y, \{x_0, ..., x_i\})$, kde funkci Reach aplikujeme přímo na původní graf $\boldsymbol{G}_0$.*

[4, str. 115]

# Kapitola 4

# Algoritmus minimálního stupně

V této kapitole, jak již její název napovídá, se podíváme na Algoritmus minimálního stupně, budeme jej zkracovat MDA (minimum degree algorithm). Jedná se o nejčastěji používáný algoritmus. Nejprve si představíme, na čem tento algoritmus stojí, poté jeho základní variantu a následně další možné způsoby rychlejší implementace a dalších případných zrychlení výpočtu. Abychom v dalším textu mohli na následující varianty MDA odkazovat, algoritmy si označíme. Značit je budeme MDA1, MDA2 a tak dále, vždy tučně, abychom odlišili, že se jedná o konkrétní algoritmus oproti obecné zkratce MDA. Algoritmus se opírá o následující myšlenku.

Nechť $\mathbf{A}$ je libovolná symetrická matice a $\mathbf{P}$ je permutační matice. Pojďme se podívat, jak je to s nulovými prvky u matic $\mathbf{A}$ a $\mathbf{PAP}^T$. Pokud matici $\mathbf{A}$ přerovnáme pomocí permutace $\mathbf{P}$, umístění nulových prvků se samozřejmě změní, ovšem jejich počet zůstane zachován. Platí $|Nonz(\mathbf{A})| = |Nonz(\mathbf{PAP}^T)|$. Nonz je zkratka pro non zeros, neboli nenulové. Co se ovšem zásadně liší, je počet nenulových prvků poté, co tyto matice rozložíme Choleského rozkladem. Tedy $|Nonz(\mathbf{F(A)})| \neq |Nonz(\mathbf{F(PAP}^T))|$.

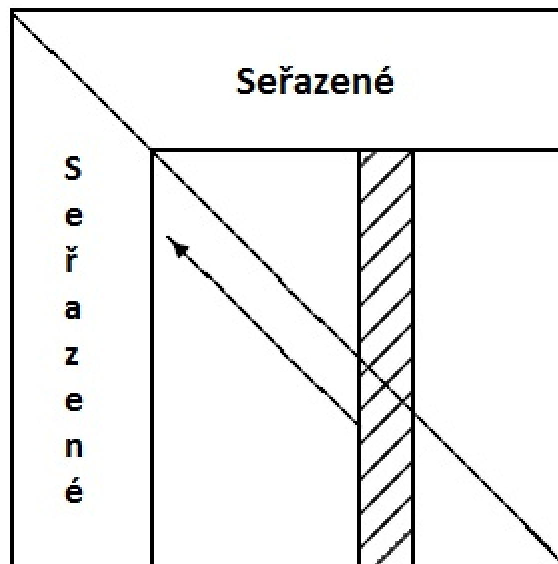
V ideálním případě bychom chtěli najít takovou permutaci $\mathbf{P}^{\star}$, která nám bude zaplňení minimalizovat, aby platilo následující:

$$|Nonz(\mathbf{F}(\mathbf{P}^{\star}\mathbf{A}\mathbf{P}^{\star^{T}}))| = \underbrace{min}_{\mathbf{P}} |Nonz(\mathbf{F}(\mathbf{P}\mathbf{A}\mathbf{P}^{T}))|.$$

[4, str. 132]

Bohužel, zatím není znám žádný efektivní algoritmus, který by nám dal optimální $\mathbf{P}^{\star}$ pro obecnou, symetrickou matici. Proto se tedy spoléháme na heuristické metody. A právě z nich je nejúspěšnější MDA.

Pokud máme již očíslovaných $\{x_1, ..., x_{i-1}\}$ uzlů, je zaplnění v těchto sloupcích již pevně dáno. Abychom minimalizovali zaplnění v $i$-tém sloupci, tak ve zbývající části matice, která má být ještě zpracována, najdeme sloupec s nejmenším počtem nenulových prvků a ten přesuneme na pozici $i$-tého sloupce. Pro názornost zobrazeno na obrázku 4.2.




Obrázek 4.1: Motivace pro algoritmus minimálního stupně.

## 4.1. Základní algoritmus

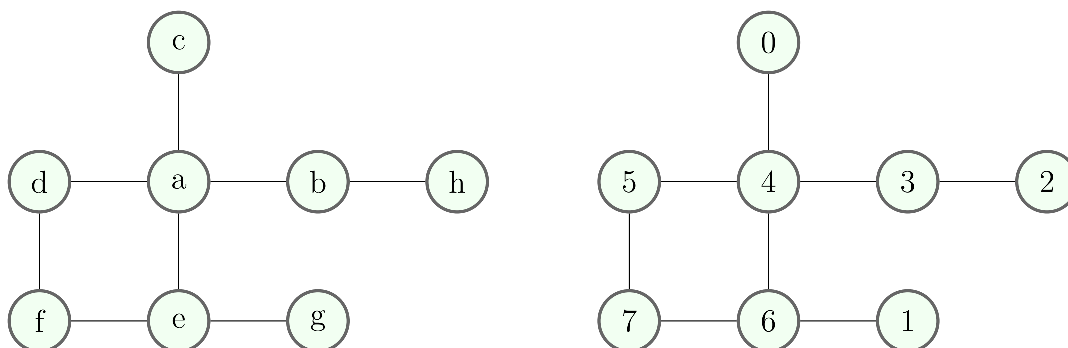
Z teoretického hlediska je základní varianta algoritmu velmi jednoduchá. Nejlépe se nám bude popisovat pomocí grafů a eliminačních grafů. Tento algoritmus si označíme jako **MDA1** a pod stejným názvem je i v přílohách. Nechť $\mathbf{G}_0 = (V, E)$ je neočíslovaný graf. Algoritmus pak lze shrnout do následujících kroků.

1. *(Inicializace)* $i = 1$.

2. *(Výběr nejmenšího stupně)* V grafu $\mathbf{G}_{i-1} = (V_{i-1}, E_{i-1})$ vyber uzel s nejmenším stupněm.

3. *(Sestavení nového grafu)* Sestav nový eliminační graf $\mathbf{G}_i = (V_i, E_i)$ tak, že z grafu $\mathbf{G}_{i-1}$ odebereš uzel $x_i$ podle věty 1.

4. *(Ukončovací kritérium)* Nastav $i = i + 1$. Pokud platí, že $i > |\mathbf{X}|$, zastav. Jinak pokračuj od kroku číslo 2.
   (Poslední krok nám tedy říká, že algoritmus pracuje v cyklu o délce $n$, kde $n$ je počet uzlů, neboli dimenze matice, dokud nevyčerpá, a tedy neočísluje všechny uzly.)




Obrázek 4.2: MDA pro graf.

Než se pustíme do úprav základního algoritmu, pojďme se podívat na použití tohoto algoritmu na konkrétním příkladu. Vezměme si matici **A**, se kterou jsme se již setkali v kapitole 3.2.

$$\mathbf{A} = \begin{bmatrix} x & x & x & x & x & 0 & 0 & 0 \\ x & x & 0 & 0 & 0 & 0 & 0 & x \\ x & 0 & x & 0 & 0 & 0 & 0 & 0 \\ x & 0 & 0 & x & 0 & x & 0 & 0 \\ x & 0 & 0 & 0 & x & x & x & 0 \\ 0 & 0 & 0 & x & x & x & 0 & 0 \\ 0 & 0 & 0 & 0 & x & 0 & x & 0 \\ 0 & x & 0 & 0 & 0 & 0 & 0 & x \end{bmatrix}.$$

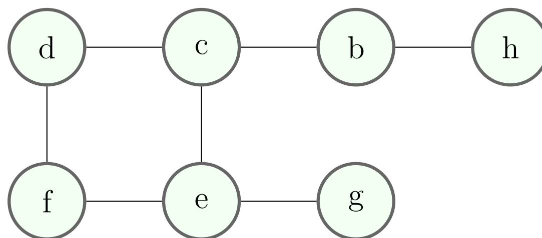
Graf matice **A** vidíme vlevo na obrázku 4.2. V prvním kroku algoritmu pracujeme tedy s celým grafem. Stupně jednotlivých uzlů jsou $deg(i) = [4, 2, 1, 2, 3, 3, 2, 1, 1]$ v pořadí od uzlu $a$ do $h$. Nejmenší stupeň uzlu je tedy číslo 1. Vidíme, že se tu vyskytuje vícekrát, ale řešení shod stupňů v základním algoritmu není nijak implementováno a vybere se tedy první v pořadí. Což je pro nás uzel $c$. Uzel $c$ tedy očíslujeme jako uzel 0. Nyní dle kroku 3 transformujeme matici. Očíslování uzlu znamená, že v matici **A** prohodíme první a třetí řádek a sloupec. Dostaneme tak matici $\mathbf{A}^P$.

$$\mathbf{A}^P = \begin{bmatrix} x & 0 & x & 0 & 0 & 0 & 0 & 0 \\ 0 & x & x & 0 & 0 & 0 & 0 & x \\ x & x & x & x & x & 0 & 0 & 0 \\ 0 & 0 & x & x & 0 & x & 0 & 0 \\ 0 & 0 & x & 0 & x & x & x & 0 \\ 0 & 0 & 0 & x & x & x & 0 & 0 \\ 0 & 0 & 0 & 0 & x & 0 & x & 0 \\ 0 & x & 0 & 0 & 0 & 0 & 0 & x \end{bmatrix}.$$

A nyní odstraníme očíslovaný uzel 0, tedy vyškrtneme první řádek a sloupec. Pracujeme tak pouze se submaticí $\mathbf{A}^1 = \mathbf{A}[1:, 1:]$, která vypadá následovně.

$$\mathbf{A}^1 = \begin{bmatrix} x & x & 0 & 0 & 0 & 0 & x \\ x & x & x & x & 0 & 0 & 0 \\ 0 & x & x & 0 & x & 0 & 0 \\ 0 & x & 0 & x & x & x & 0 \\ 0 & 0 & x & x & x & 0 & 0 \\ 0 & 0 & 0 & x & 0 & x & 0 \\ x & 0 & 0 & 0 & 0 & 0 & x \end{bmatrix}.$$

Této matici $\mathbf{A}^1$ odpovídá graf $\mathbf{G}_1 = (V_1, E_1)$.




Stupně jednotlivých uzlů jsou $deg(i) = [2, 3, 2, 3, 2, 1, 1]$ v pořadí uzlů od $b$ do $h$. A nyní bychom zase vybrali uzel s nejmenším stupněm a takto dál, dokud bychom neočíslovali všechny uzly v grafu. Výsledný graf lze vidět vpravo na obrázku 4.2.

## 4.2. MDA pomocí dosažitelných množin

Použití posloupností grafů v základním algoritmu nám dává do rukou první použitelný nástroj k určení uzlu, který bude očíslován jako další. A tedy k vhodnému přerovnání matice. Nicméně z hlediska implementace je sestavení nového grafu nejnáročnějším krokem celého algoritmu. Vyplynula tak otázka, zdali není možnost pracovat pouze s originálním grafem $\mathbf{G}_0 = (V_0, E_0)$. Tedy na něm určit celé očíslování grafu, nikoli vždy jen další krok a pak sestavit graf nový. Takto bychom se totiž vyhnuli tvorbě posloupnosti grafů a tím celý algoritmus podstatně zrychlili. Odpověď je samozřejmě kladná, a na otázku, jak to udělat, nám dá odpověď funkce *Reach*, se kterou jsme se již seznámili v kapitole 3.4. Variantu Algoritmu minimálního stupně s použitím funkce *Reach* si označme jako **MDA2**. Pomocí *Reach* lze základní algoritmus popsat takto.

1. *(Inicializace)* $S = \emptyset$. $Deg(x) = |Adj(x)|$ pro $x \in V$.

2. *(Výběr nejmenšího stupně)* Z množiny $V - S$ vyber uzel $y$ s nejmenším stupněm. Tento uzel očísluj a nastav $T = S \cup \{y\}$.

3. *(Aktualizace stupňů)* $Deg(u) = |Reach(u, T)|$ pro $u \in X - T$.

4. *(Ukončovací kritérium)* Nastav $S = T$. Pokud platí $T = X$, zastav. Jinak pokračuj od kroku číslo 2.

Tímto přístupem využíváme pouze původní graf $\mathbf{G}_0$, konkrétně strukturu seznamů sousedů, a to po celou dobu běhu algoritmu. Právě na této struktuře v kroku 3, pomocí funkce *Reach*, vždy přepočítáváme stupně uzlů spolu s tím, jak jednotlivé uzly očíslováváme (eliminujeme).

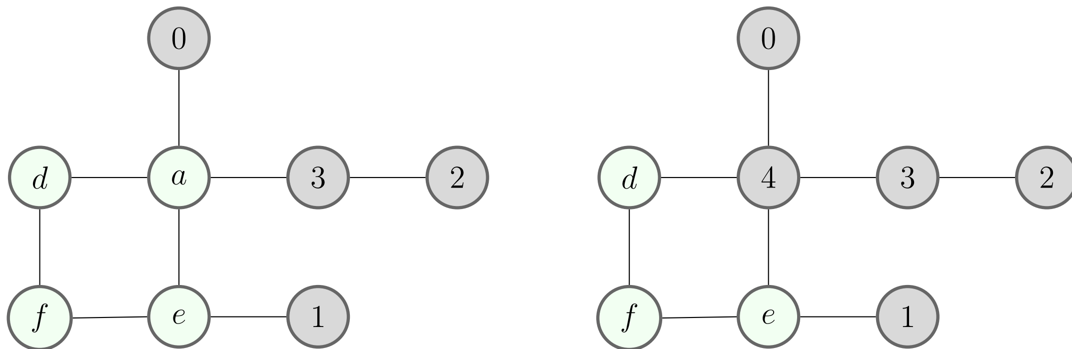
Množina $S$ nám zde říká, které uzly jsme již zpracovali a očíslovali. Na začátku je prázná a postupně nám narůstá. V druhém kroku vybereme uzel s nejmenším stupněm z uzlů, které jsou doposud neočíslované, tedy nepatří do $S$. Ovšem po eliminaci uzlu se nám stupně změní, což musíme zohlednit, a proto přepočítáváme stupně. V základní verzi jsme sestavovali nový graf, zde jen aplikujeme funkci *Reach* na původní graf a tím nám odpadá nutnost sestavovat nové grafy. V posledním kroku řešíme pouze, kdy algoritmus zastavit, což nastane v případě, že jsme prošli včechny uzly v cyklu $n$.

Nicméně, pokud se nad algoritmem ještě dále zamyslíme, zjistíme, že není nutno v každém kroku přepočítávat všechny uzly z množiny $X - T$. Stupeň se bude měnit pouze u části z nich. Pokud totiž eliminujeme uzel $y$, mění se nám pouze stupně uzlů, jež patří do $Adj(y)$ v daném kroku eliminace. Jako vždy si to pojďme ukázat na příkladu. Vezměme si zase náš již několikrát používaný graf $\mathbf{G}_0$ (4.2 vlevo). A podívejme se na krok, ve kterém budeme očíslovávat uzel $a$. Nyní již máme uzly $c, g, h, b$ (v tomto pořadí) očíslované jako 0, 1, 2, 3 (znázorněno šedou výplní na obrázku 4.3 vlevo). Tyto uzly nyní patří do množiny $S_4$. Uzel $a$ očíslujeme, tedy $a = x_4$. Množinu $T$ nastavíme jako $T = S + \{a\}$ (obr. 4.3 vpravo). Celkově tak máme:

$$S_4 = \{x_0, x_1, x_2, x_3\},\ T = \{x_0, x_1, x_2, x_3, x_4\}.$$

Platí $Reach(a, S_3) = \{d, e\}$. Tedy v kroku 3 našeho algoritmu stačí přepočítat stupně uzlů $d$ a $e$.

$$Deg(d) = Reach(d, T) = |\{e, f\}| = 2 \text{ a } Deg(e) = Reach(e, T) = |\{d, f\}| = 2.$$




Obrázek 4.3: Reach a MDA

Můžeme tak krok 3 v našem algoritmu přepsat následovně.
*(Aktualizace stupňů)* $Deg(u) = |Reach(u, T)|$ pro $u \in Reach(y, S)$.

## 4.3. Moje implementace MDA

V této sekci bych rád shrnul a popsal svoji implementaci Algoritmu minimálního stupně, sepsanou v Python 3.6. Důvodem, proč ji zde uvádím, je, že při testování na mnou generovaných náhodných maticích vykazuje ještě menší čas výpočtu, než algoritmus s užitím funkce *Reach*. Časy měření a další příklady následují v další kapitole. Než jsem se dostal k verzi algoritmu, kterou zde uvedu, proběhlo mnoho úprav. Vycházel jsem z algoritmů **MDA1** a **MDA2** a postupně je zdokonaloval se snahou o zrychlení výpočtu. Vstupem pro tento algoritmus, který si zde označíme jako **MDAGtoP**, je graf dané matice **A**. Výstupem je permutační matice **P**. Po provedení permutace $\mathbf{PAP}^T$ dostaneme stejný výsledek jako algoritmus **MDA2**. Nyní tedy již k samotné implementaci.

1. *(Inicializace)*
   Vstupem je $graf$.
   $\mathbf{P} = zeros(n, n)$,
   $deg(x) = zeros(n)$,
   $update = 0, 1, ..., n$,

$j = 0$.

Vstupem je graf, označme jej jako $graf$. Jedná se o seznam o délce $n$, kde každá položka je $Adj(x_i)$ (tedy další seznam). Matice $\mathbf{P}$ je naše výsledná permutační matice, ve které zaznamenáváme prohazování uzlů. V seznamu $deg(x)$ budeme mít stupně jednotlivých uzlů a množina *update* nám slouží k tomu, abychom přepočítávali jen potřebné uzly, nikoliv všechny. Přes $j$ budeme projíždět cyklus o délce $n$.

2. *(Aktualizace stupňů a výběr nejmenšího stupně)*

   $deg(x_i) = |Adj(x_i)|$, pro $i \in update$.

   $mini = argmin(deg)$,

   $P(j, mini) = 1$.

   V prvním kroku zde napočítáme stupně uzlů. Ty zjistíme jen jako délku seznamu $Adj(x)$ a v dalších krocích už pak přepočítáváme pouze ty uzly, jež jsou obsaženy v množině *update*. Pak vybereme uzel s nejmenším stupněm (shody opět neřešíme a vybereme první v pořadí). *mini* tedy obsahuje číslo uzlu v původním uspořádání, který má nejmenší stupeň v tomto kroku algoritmu. Tento uzel bude očíslován jako $j$. Očíslování zapíšeme do permutační matice.

3. *(Transformace grafu)*

   $update = Adj(mini)$,

   $deg(mini) = n + 1$,

   $graf = eliminuj(mini, graf)$,

   $graf[mini] =$ "*hotovo*".

   V tomto kroku si do *update* uložíme sousedy uzlu, který vyeliminujeme. Jiné totiž přepočítávat nepotřebujeme. Stupeň vyeliminovaného uzlu nastavíme na $n + 1$, tedy větší, než maximální možný stupeň, abychom zabránili jeho opětovnému výběru. Neodebíráme jej úplně, neboť takto nemusíme upravovat indexy ostatních uzlů. Kdybychom uzel odebrali, museli bychom přeindexovat všechny ostatní uzly za odebraným uzlem a nebo mít další se-

znam pro indexy uzlů. Dále graf v tomto kroku přetransformujeme pomocí pomocné funkce *eliminuj*. Ta funguje následovně.

Vstupem je uzel, který chceme symbolicky vyeliminovat, a graf, ve kterém budeme eliminaci provádět. Vezme $Adj(mini)$ a ke každému uzlu $x_i$ z této množiny přidá k $Adj(x_i)$ všechny ostatní uzly z $Adj(mini)$, kromě uzlu $x_i$, které v této množině $Adj(x_i)$ doposud nejsou. Jedná se vlastně o symbolické vyeliminování uzlu *mini* a tím dochází k redukci grafu v dalších krocích. Vyeliminovaný uzel totiž označíme "hotovo"a dále už s ním nijak nepracujeme. Opět jej ze seznamu neodebereme úplně, protože by se nám změnila délka pole a tím i indexy ostatních uzlů.

4. *(Ukončovací kritérium)*

   Nastav $j = j + 1$. Pokud platí $j = n$, zastav, neboť jsme očíslovali všechny uzly. Jinak pokračuj od kroku číslo 2.

   Tím je algoritmus u konce. Výsledkem je permutační matice **P**.

# Kapitola 5

# Příklady a testování

V této kapitole si ukážeme hned několik věcí týkajících se MDA. Předně, jak krok po kroku využít algoritmus při řešení soustav lineárních rovnic na malých maticích. Důvod, proč začínáme právě menšími maticemi, je, abychom je zde mohli mít explicitně uvedené. Postup je samozřejmě stejný i na velkých maticích, kde, jak již čtenář ví, je jeho použití bezesporu výhodnější. Následně si ukážeme, jak bude vypadat přerovnání matic velkých. Ty už zde explicitně psát nebudeme kvůli jejich rozměrům. Nicméně si uvedeme alespoň náhledy rozmístění prvků. Třetím bodem bude měřit zaplnění na maticích různých velikostí mezi jednotlivými variantami algoritmu. Uvedeme si také průměrné doby výpočtů jednotlivých algoritmů na maticích různých velikostí a také srovnání s případem, kdy MDA používat nebudeme. Všechny algoritmy, které zde zmíníme, jsou popsány v následující kapitole. Proto je zde nebudeme nijak detailněji popisovat.

## 5.1. Příklad na řešení soustavy rovnic

Jak tedy vyřešit následující soustavu rovnic? Matici levé strany budeme značit **A** a velktor pravé strany **b**.

$$
\begin{aligned}
+3x_1 - x_2 \quad +2x_3 \qquad\qquad\qquad + 2x_6 \qquad\qquad\qquad &= 19 \\
-x_1 + 7x_2 \qquad\qquad\qquad +3x_5 + 2x_6 \qquad - x_8 \quad &= 32 \\
+2x_1 \qquad +3x_3 \qquad\qquad\qquad\qquad\qquad\qquad\qquad &= 11 \\
+ 7x_4 \qquad + x_6 \quad +4x_7 \qquad\quad &= 62 \\
+ 3x_2 \qquad\qquad\qquad +5x_5 \qquad -x_7 \qquad\quad &= 24 \\
+2x_1 + 2x_2 \qquad + x_4 \qquad + 6x_6 \qquad\qquad\qquad &= 46 \\
+ 4x_4 \quad -x_5 \qquad +6x_7 \qquad\quad &= 53 \\
- x_2 \qquad\qquad\qquad\qquad\qquad\qquad\qquad + 2x_8 \quad &= 14,
\end{aligned}
\tag{5.1}
$$

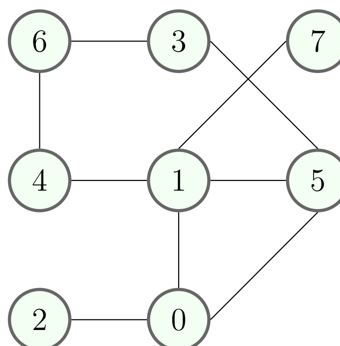
V souladu se schématem popsaným v sekci 3.1 budeme mít tuto soustavu v počítači zapsanou v CCS formě následovně.

$x = [0, 1, 2, 5, 0, 1, 4, 5, 7, 0, 2, 3, 5, 6, 1, 4, 6, 0, 1, 3, 5, 3, 4, 6, 1, 7]$,

$y = [0, 4, 9, 11, 14, 17, 21, 24, 26]$,

$hodnoty = [3, -1, 2, 2, -1, 7, 3, 2, -1, 2, 3, 7, 1, 4, 3, 5, -1, 2, 2, 1, 6, 4, -1, 6, -1, 2]$,

Pro užití MDA, musíme mít graf. Začneme tím, že sestavíme z této řídké matice graf, který vypadá následovně.




Nyní můžeme na tento graf aplikovat algoritmus minimálního stupně, abychom jej přečíslovali. Použijeme tedy **MDAGtoP** ze sekce 4.3. výsledkem je matice

$$\mathbf{P} = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}.$$

Tedy můžeme soustavu rovnic přerovnat $\mathbf{A}^2 = \mathbf{PAP}^T$. Spolu s ní přerovnáme i vektor $\mathbf{b}^2 = \mathbf{Pb}$. Vzhledem k tomu, že je matice symetrická, bude nám pro Choleského rozklad stačit vzít dolní polovinu přerovnané matice $\mathbf{A}^2$. Ta, zapsaná v CSS, vypadá následovně.

$x = [0, 2, 1, 3, 2, 3, 6, 3, 5, 6, 4, 6, 7, 5, 7, 6, 7]$,

$y = [0, 2, 4, 7, 10, 13, 15, 16, 17]$,

$hodnoty = [3, 2, 2, -1, 3, -1, 2, 7, 3, 2, 7, 1, 4, 5, -1, 6, 6]$.

Právě při provádění Choleského rozkladu na řídké struktuře ušetříme obrovské množství času a paměti při výpočtech. Pokud totiž provádíme Choleského rozklad na husté matici, jak jsme si jej popsali v sekci 2.2, provádíme řádově $n^3$ operací, což je pro velké matice enormní množství. Zde, při použití Choleského rozkladu pro řešení soustavy s řídkou maticí ušetříme nejvíce času. Přesto je Choleského rozklad výpočetně nejnáročnější částí řešení soustavy s řídkou maticí. Při testování pro dimenzi $n = 2000$ mi Choleského rozklad zabral 41% celkového času výpočtu, pro dimenzi $n = 5000$ to bylo 61%. Pro $n = 8000$ to bylo už dokonce 69%. Je tedy vidět rostoucí trend. Choleského rozklad je tak nejnáročnější částí řešení soustavy i pro řídké matice. Je možné, že můj kód pro Choleského rozklad by se dal zefektivnit, nicméně netuším zatím jak, a bylo by třeba tomu věnovat více času. Srovnání rozkladu pro řídké a pro husté si ukážeme v sekci měření.

Zpět k řešení našeho příkladu. Po provedení Choleského rozkladu dostaneme matici **L**. Modifikaci Choleského rozkladu mám ve svých algoritmech označenou jako **Cholezskysparse**. **L** bude zase v CCS formě. Zde ji ovšem uvedu v husté

formě kvůli přehlednosti. Navíc ji zaokrouhlíme na 3 desetinná místa.

$$\mathbf{L} = \begin{bmatrix} 1.732 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1.414 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1.155 & 0 & 1.291 & 0 & 0 & 0 & 0 & 0 \\ 0 & -0.707 & -0.775 & 2.429 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2.646 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1.235 & 0 & 1.864 & 0 & 0 \\ 0 & 0 & 1.549 & 1.317 & 0.378 & -0.873 & 0.980 & 0 \\ 0 & 0 & 0 & 0 & 1.512 & -0.536 & -1.061 & 1.517 \end{bmatrix}.$$

Nyní už jen vyřešíme soustavu $\mathbf{Lc} = \mathbf{b}$ a následně soustavu $\mathbf{L}^T\mathbf{x} = \mathbf{c}$. Oba tyto kroky řeší algoritmus označený jako **RRS**. Výsledkem je tak vektor $\mathbf{x}^2 = [3, 8, 1, 2, 4, 5, 6, 7]^T$. Ten je řešením naší přerovnané soustavy $\mathbf{A}^2\mathbf{x}^2 = \mathbf{b}^2$. Řešením původní soustavy $\mathbf{Ax{=}b}$ je vektor $\mathbf{x} = [1, 2, 3, 4, 5, 6, 7, 8]^T$, který dostaneme jako $\mathbf{x} = \mathbf{x}^2\mathbf{P}$.

## 5.2. testování

Nyní se pojďme podívat, jak přerovnávání matic pomocí Algoritmu minimálního stupně ovlivní Choleského rozklad u velkých řídkých matic. Tato část se bude skládat především z obrázků, protože zde se budeme zabývat pouze rozmístěním prvků ve velkých maticích. Tedy jejich symbolickou strukturou. Každý obrázek obsahuje 4 grafy.

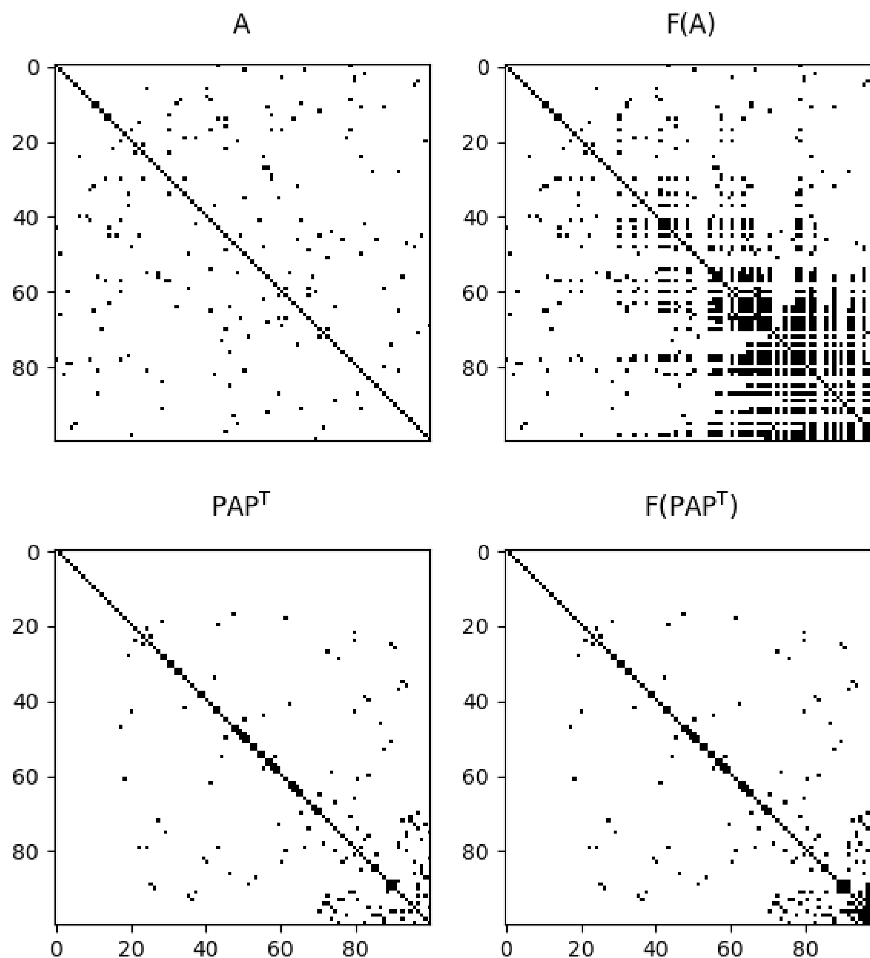
V levém horním rohu je původní matice **A**, náhodně vygenerovaná.

Vpravo od ní je její matice zaplnění $\mathbf{F(A)}$. Pro připomenutí $\mathbf{F(A){=}L{+}L}^T$, kde **L** je Choleského rozklad matice **A**.

Vlevo dole je matice $\mathbf{PAP}^T$, tedy matice přerovnaná pomocí MDA.

A vpravo od ní je její matice zaplnění $\mathbf{F(PAP}^T\mathbf{)}$

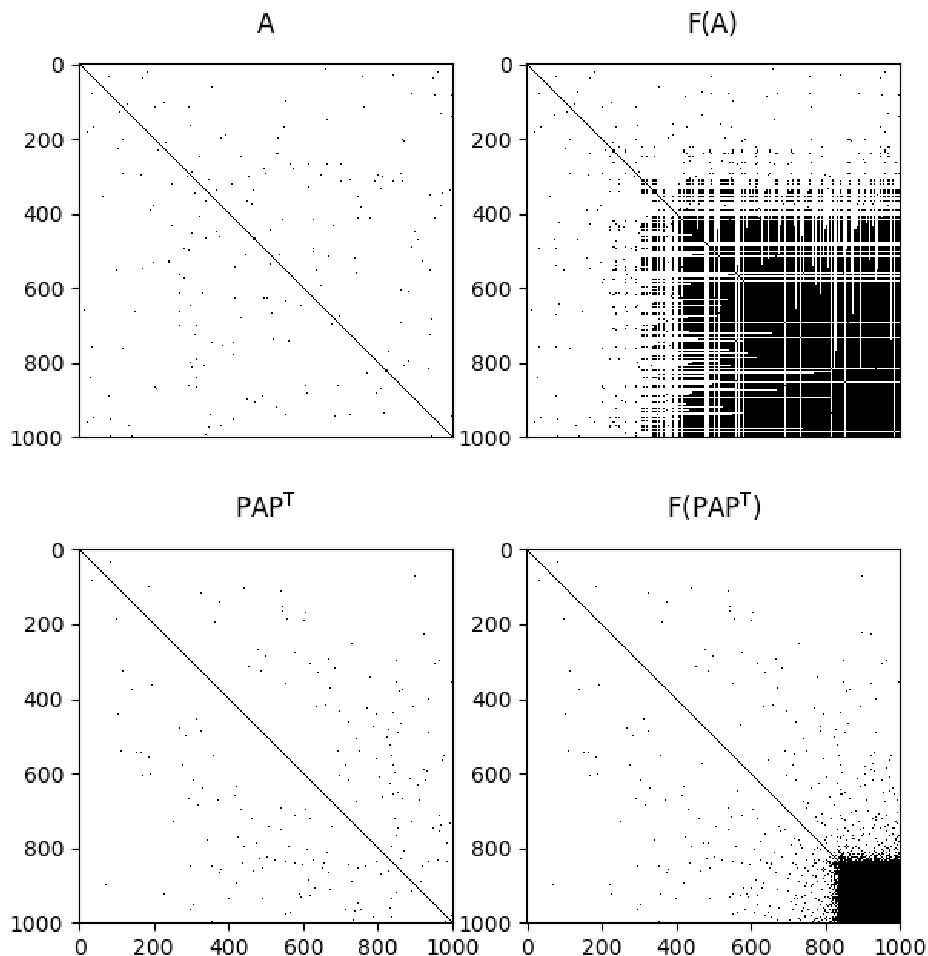
Začneme tedy maticí s rozměry $n = 100$. Data k této konkrétní matici jsou v přílohách označena "$PR2_1$". Zde platí, že původní matice $\mathbf{A}$ má 280 prvků.




Obrázek 5.1: MDA pro $n = 100$

Matice $\mathbf{PAP}^T$ má tedy stejně. Ovšem $\mathbf{F(A)}$ má 1 234 prvků. Zato $\mathbf{F(PAP}^T\mathbf{)}$ jich má pouze 332.

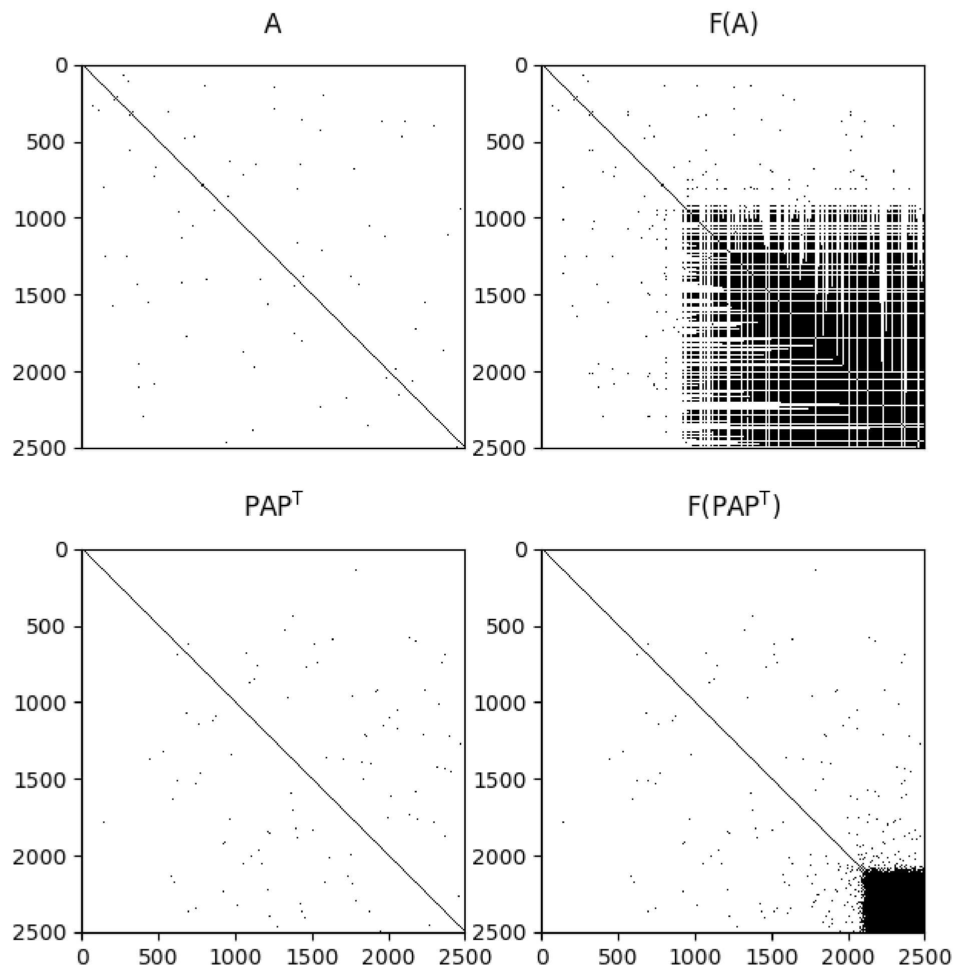
Nyní si zvolíme rozměry $n = 1000$. Data k této konkrétní matici jsou v přílohách označena "$PR2_2$". Zde platí, že původní matice $\mathbf{A}$ má 4 372 prvků.




Obrázek 5.2: MDA pro $n = 1000$

Matice $\mathbf{PAP}^T$ má tedy stejně. Ovšem $\mathbf{F(A)}$ má 250 692 prvků. Zato $\mathbf{F(PAP}^T\mathbf{)}$ jich má pouze 38 840.

Na závěr se podívejme na rozměry $n = 2500$. Data k této konkrétní matici jsou v přílohách označena “$PR2_3$“. Zde platí, že původní matice $\mathbf{A}$ má 10 632




Obrázek 5.3: MDA pro $n = 2500$

prvků. Matice $\mathbf{PAP}^T$ má tedy stejně. Ovšem $\mathbf{F(A)}$ má 1 301 472 prvků. Zato $\mathbf{F(PAP}^T\mathbf{)}$ jich má 193 174.

Do vyšších rozměrů se zde pouštět nebudeme, hlavně z důvodu problematiky zobrazení. Už při rozměru $n = 2500$ lze stěží vidět prvky matice $\mathbf{A}$. Navíc i zde je v matici s $n = 1000$ a $n = 2500$ nastavená dvojnásobná hustota oproti $n = 100$, aby byly znatelné alespoň nějaké prvky. Trend MDA zde jde vidět krásně i přesto. Ve všech případech se snaží prvky shluknout do pravého dolního rohu, kde následný Choleského rozklad vytvoří menší zaplnění.

V následujících řádcích si popíšeme mnou prováděná měření. Vždy jsem prováděl 250 měření, postupně pro dimenze $n = 100, 1000, 2500$. Měřil jsem 2 typy řídkých matic. První, řidší, měla zaplnění $2/n$ a druhá, dvakrát hustší, měla zaplnění zhruba $4/n$. U těchto matic jsem srovnával rozdíl jejich zaplnění po provedení Choleského rozkladu při použití MDA a bez něj.

Pro $n = 100$ bylo naměřeno průměrně $9\times$ vyšší zaplnění při nepoužití MDA. U hustějších matic to bylo pouze $5.21\times$ větší zaplnění.

Pro $n = 1000$ se průměrně zaplnilo už $14.15\times$ více míst při výpočtu Choelzského rozkladu bez použití MDA. Při měření na hustší matici to bylo už jen $6.32\times$ více. A pro $n = 2500$ nám vyšlo zaplnění bez MDA $17.7\times$ vyšší. A $7.07\times$ pro více zaplněné matice.

Lze pozorovat, že čím je hustší původní matice $\mathbf{A}$, tím méně je MDA výhodné, což jsme si řekli už dříve a nyní jen potvrdili pomocí výpočtů. Ale stejně tak čím vyšší rozměry matic máme, tím znatelnější je úspora času.

Lze si také všimnout, že od jistého $i$ je obecně submatice $\mathbf{F(A)}_{[i:,i:]}$ hustá. A MDA na husté matici je velice neefektivní, tedy zajímavou variantou pro zrychlení výpočtu by mohlo být neprovádět MDA až do vyčerpání všech uzlů ale jen pro prvních 80% uzlů. Tato hranice se však může u jednotlivých matic velice lišit.

Jak bylo zmíněno v úvodu této kapitoly, následuje měření délky trvání výpočtů u jednotlivých variant MDA pro různé $n$. Výsledky shrneme do následujících tabulek. Časy jsou uvedené v sekundách, $n$ jako vždy vyjadřuje velikost matice a $i$ je počet provedených měření, ze kterých byl spočítán průměrný čas. Přičemž v každém kroku byla vygenerovaná nová náhodná matice, na kterou byly aplikovány všechny algoritmy, které takto běžely na těch samých maticích.

Tabulka 5.1: $n = 100, i = 500$

| **Algoritmus** | **Čas výpočtu** |
|---|---:|
| MDA1 | 0.176 |
| MDA2 | 0.065 |
| MDAGtoP | 0.007 |

Tabulka 5.2: $n = 200, i = 500$

| **Algoritmus** | **Čas výpočtu** |
|---|---:|
| MDA1 | 1.133 |
| MDA2 | 0.318 |
| MDAGtoP | 0.023 |

Tabulka 5.3: $n = 400, i = 500$

| **Algoritmus** | **Čas výpočtu** |
|---|---:|
| MDA1 | 8.833 |
| MDA2 | 2.422 |
| MDAGtoP | 0.087 |

Už zde jde vidět, že MDA1 za zbývajícími algoritmy velmi zaostává. Je to způsobeno tím, že sestavování grafu v každém kroku je velmi náročné, obzvláště z matice zapsané v dvourozměrném poli. Složitost algoritmu se tak pohybuje řádově kolem $n^3$, což je pro velké matice nepřijatelné.

Taktéž pozorujeme, o kolik je MDA2 efektivnější oproti MDA1. Funkce *Reach* totiž pracuje mnohem rychleji, než sestavování celého nového grafu. Nicméně, jak uvidíme v následujících tabulkách, u vyšších $n$ začne i tato funkce zaostávat. Neboť procházet velkou množinu $S$, uzel po uzlu, je náročné. Nyní už tedy z tabulky vypustíme algoritmus MDA1.

Tabulka 5.4: $n = 800, i = 500$

| **Algoritmus** | **Čas výpočtu** |
|---|---|
| MDA2 | 24.578 |
| MDAGtoP | 0.348 |

Tabulka 5.5: $n = 1600, i = 200$

| **Algoritmus** | **Čas výpočtu** |
|---|---|
| MDA2 | 324.241 |
| MDAGtoP | 1.583 |

Jde tedy vidět, že pro narůstající $n$ je i MDA2 nedostatečný. Je možné, že je to mou implementací funkce *Reach*, protože ta u vyšších $n$ trvá velmi dlouho. Nicméně pozorujeme, že MDAGtoP má stále velmi nízký čas výpočtu. V následující tabulce si ukážeme jeho dobu trvání na ještě vyšších rozměrech.

Tabulka 5.6: MDAGtoP pro různá $n$, $i = 200$

| **n** | **Čas výpočtu** |
|---|---|
| 2000 | 2.526 |
| 3000 | 6.712 |
| 4000 | 12.639 |
| 5000 | 23.179 |
| 6000 | 30.078 |
| 8000 | 66.636 |

Tabulka ukazuje, že s časem okolo minuty tento algoritmus zvládne přerovnat

i matici o rozměrech $n = 8000$, což je zcela jistě obrovský rozdíl oproti dvěma předchozím algoritmům.

Nyní, na závěr této kapitoly, se pojďme podívat, jaký bude rozdíl v času výpočtu řešení soustavy rovnic při použití MDA, oproti případu, kdy jej nevyužijeme. Opět vygenerujeme náhodnou matici $\mathbf{A}$ a k ní pravou stranu $\mathbf{b}$. Poprvé tuto soustavu vyřešíme s použitím algoritmu MDAGtoP a posléze vyřešíme úplně stejnou soustavu pouze za použití Choleského rozkladu, tak jak jsme si jej popsali v sekci 2.3. Do tabulky opět zapíšeme průměrné časy výpočtů, získané z $i$ měření.

Tabulka 5.7: Časy řešení soustav pro různá $n$, $i = 200$

| **n** | **bez MDA** | **s MDA** |
|---|---:|---|
| 10 | 0.0005 | 0.003 |
| 20 | 0.001 | 0.007 |
| 50 | 0.015 | 0.021 |
| 60 | 0.027 | 0.027 |
| 100 | 0.104 | 0.052 |
| 200 | 0.754 | 0.169 |
| 500 | 10.957 | 0.711 |
| 1000 | 78.979 | 2.470 |
| 2500 | 1 243.429 | 16.971 |
| 5000 | 10 134.558 | 96.118 |

Data z této tabulky zřetelně ukazují na efektivitu MDA oproti běžnému Choleskému rozkladu. Pro $n < 60$ je sice Choleského rozklad rychlejší, nicméně pouze v zanedbatelném měřítku. Zajímají nás velká $n$ a pro ty MDA dává časy dokonce řádově rychlejší. Pro $n = 5000$ výpočet trvá minutu a půl při použití MDA, ale bez něj nám Choleského rozklad zabere bezmála 3 hodiny. Zde by bylo dobré podotknout, že průměrný čas pro $n = 5000$ bez MDA byl spočítán pouze z deseti měření, nikoli z dvěstě, a to kvůli časové náročnosti.

Na závěr této kapitoly bych rád připomněl, že časy výpočtů na různých počítačích se liší dle jejich výkonu. Tedy na jiném počítači nejspíše budou výsledky odlišné, ale poměry mezi výsledky by měly zůstat zachovány.

# Kapitola 6

# Algoritmy

Tato kapitola obsahuje seznam a krátký popis algoritmů, které jsou obsaženy v přílohách. Jedná se tedy o různé algoritmy, ať už vetší jako MDA nebo jen malé pomocné funkce, které se v této práci vyskytly.

1. *MDA1*

   Základní Algoritmus minimálního stupně. Dopodrobna popsaný v 4.1.

2. *MDA2*

   Upravený Algoritmus minimálního stupně pomocí funkce *Reach*, opět dopodrobna popsaný v 4.2.

3. *MDAGtoP*

   Moje implementace Algoritmu minimálního stupně. Popsaná v 4.3.

4. *genPDM*

   Generuje náhodnou pozitivně definitní, řídkou matici. Řídkost matice je $2/n$. Pozitivní definitnost zajištěna tím, že se na diagonálu přičte součet prvků v daném řádku. Mimo diagonálu se vyskytují náhodná celá čísla v rozmezí od -1 do 5.

5. *densetograph*

   Tato funkce sestaví graf z matice ve dvourozměrném poli. Řádově $n^2$ operací.

6. *sparsetograph*

   Tahle sestaví graf z matice v CCS zápisu. Řádově $p$ operací, kde $p$ je počet nenulových prvků matice. Pracuje tedy rychleji než *densetograph*.

7. *sparsetodense*

   Slouží k převodu z CCS formy do dvourozměrného pole.

8. *densetosparse*

   Slouží k převodu z dvourozměrného pole do CCS formy.

9. *DtSL a DtSU*

   Obě slouží opět k převodu z dvourozměrného pole do CCS formy. Nicméně $DtSL$ vezme pouze dolní polovinu matice a $DtSU$ pouze horní polovinu matice. Slouží tedy převážně k převodu matice $\mathbf{L}$ a $\mathbf{L}^T$ do CCS formy. Pracují totiž jen s polovinou matice a jsou tedy rychlejší.

10. *eliminuj*

    Drobná funkce, která operuje na grafu. Z daného grafu nám vyeliminuje zadaný uzel tak, že ke všem sousedům eliminovaného uzlu doplní ostatní sousedy eliminovaného uzlu. Pracuje tedy s rychlostí $s^2$, kde $s$ je počet sousedů eliminovaného uzlu.

11. *Cholezskydense*

    Jedná se o implementaci Choelzského rozkladu tak, jak jsme si jej popsali v sekci 2.3.

12. *Cholezskysparse*

    Tohle je Choleského rozklad na CCS struktuře. Vstupem je dolní trojúhelníková část matice, jíž chceme rozložit, zapsaná v CCS formě a graf celé této matice. Graf kvůli symbolické faktorizaci, která se zde taktéž provádí. Tedy tento algoritmus operuje na struktuře, která je výstupem z algoritmu $DtSL$.

13. *Reach*

    Vstupem pro tuto funkci je uzel $x_i$, množina $S$ a graf. Pracuje tak, že na

začátku do seznamu *zbyva* dá sousedy zadaného uzlu $s_i$. Poté tento seznam prochází. Pokud vybraný uzel $x_j$ ze seznamu *zbyva* ještě nebyl zpracován, funkce zkontroluje, zdali tento uzel patří do $S$ nebo nikoli. Pokud patří do $S$, tento uzel $x_j$ je přidán do seznamu *zbyva* na konec a po čase jej funkce prochází dále, až se k němu v seznamu dopracuje. Nepatří-li uzel $x_j$ do $S$, pak se jedná o uzel, který patří do dosahu uzlu $x_i$ a tedy jej funkce připíše do seznamu *dosah*, který je výstupem této funkce. Takto funkce pokračuje, dokud seznam *zbyva* není prázdný.

14. *AdtoLPsparse*

    Jedná se o shrnutí několika již zmíněných funkcí pro zpřehlednění praktických výpočtů. Tato funkce dostane na vstupu matici $\mathbf{A}$, již jsme si náhodně vygenerovali pomocí *genPDM*. Tuto matici převede do CCS formy, provede MDA, přerovná matici, a poté provede její Choleského rozklad na CCS formě pomocí *Cholezskysparse*. Výstupem je tedy matice $\mathbf{L}$, pro přehlednost na konci této funkce převedená do standartního zápisu pomocí dvourozměrného pole, nikoliv v CCS formě.

15. *RRS*

    Funkce, která řeší soustavu rovnic v CCS zápisu. Vstupem je tedy CSS zápis matice $\mathbf{L}$ a matice $\mathbf{L}^T$ a vektor pravé strany $\mathbf{b}$. Výstupem je vektor $\mathbf{x}$ jakožto řešení této soustavy $\mathbf{Ax}=\mathbf{b}$.

16. *Priklad1*

    Zpracovaný příklad ze sekce 5.1.

17. *Priklad2*

    Zpracovaný příklad ze začátku sekce 5.2, kde jsme srovnávali zaplnění. Na začátku tohoto skriptu se načtou data dle požadované matice. Pokud chceme obrázek 5.1, načteme data "*PR2_1.npy*", pokud chceme obrázek 5.2, načteme data "*PR2_2.npy*" a pro obrázek 5.3 načteme data "*PR2_3.npy*".

18. *Priklad3*

    Příklad na měření času pro jednotlivé MDA, ve kterém měříme časy jejich trvání. Na začátku pouze nastavíme $n$ na požadovanou hodnotu a spustíme skript. Postupně pro MDA, MDA2 a MDAGtoP dostaneme výsledky $v1$, $v2$ a $v3$.

19. *Priklad4*

    Poslední příklad slouží k měření časové náročnosti výpočtu řešení soustav. Srovnáváme zde řešení soustavy Choleského rozkladem tak, jak jsme si jej popsali v sekci 2.3, s řešením soustavy pomocí MDA tak, že pracujeme s CCS formou matice.

# Závěr

V této práci jsme si představili Algoritmus minimálního stupně, jeho motivaci, základní variantu a další úpravy. Tomu předcházelo pochopení důležitých vlastností a jevů pro řídké matice, jako je například CCS forma pro ukládání matic do paměti, zaplňování a pochopení souvislosti mezi grafy a maticemi. Všem těmto poznatkům jsme se věnovali v kapitole 3. V kapitole 2 jsme si zopakovali, jak řešit rovnice, a poté jsme si na příkladu 5.1 ukázali, jak při řešení rovnic využít MDA krok za krokem. V dalších příkladech z 5. kapitoly jsme prováděli testování a výpočetní experimenty, nakolik se nám podařilo cíl splnit. Výsledky ukazují, že již i pro malé matice ($n > 100$) se MDA vyplatí. Taktéž lze vidět, že samotný MDA pracuje velmi rychle i bez dalších vylepšení, která jsou možná. Mezi tyto vylepšení patří ku příkladu masová eliminace a další varianty, o kterých se lze dočíst v [3], ale které jsem zde z časových důvodů nerozebíral. Místo toho jsem se vydal cestou implementace vlastního algoritmu, který dává výsledky rychleji, než varianta MDA2. Ačkoli mám ještě pár nápadů, jak svůj algoritmus zlepšit, nezbyl mi k tomu již bohužel čas. I přesto věřím, že byla prokázána znalost dané tématiky. Pokud srovnáme časy z testů v sekci 5.2, lze vidět, že stále nejnáročnějším krokem při řešení soustavy je Choleského rozklad, a to i přesto, když jej provádíme na CCS formě matice. Nicméně i přesto je užití MDA mnohem rychlejší, než obyčejný Choleského rozklad na matici husté.

# Literatura


[1] Davis T. A.: *Direct Methods for Sparse Linear Systems*, University of Florida, 2006.

[2] Pissanetzky S.: *Sparse Matrix Technology*, ACADEMIC PRESS, 1984.

[3] Meurant G.: *Gaussian Elimination for the Solution of Linear Systems of Equations*, Elsevier Science B.V., 2000.

[4] George A., Liu J., Ng E.: *Computer solution of sparse linear system*, 1994.

[5] Fiedler M.: *Speciální matice a jejich použití v numerické matematice*, Nakladatelství technické literatury, Praha, 1981.

[6] Segethová J.: *Základy numerické matematiky*, Karolinum, 2002.