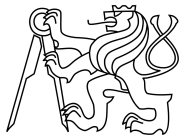
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Efektivní LU rozklad pro řídké matice
Student:	Lukáš Jurásek
Vedoucí:	doc. Ing. Ivan Šimeček, Ph.D.
Studijní program:	Informatika
Studijní obor:	Teoretická informatika
Katedra:	Katedra teoretické informatiky
Platnost zadání:	Do konce letního semestru 2017/18

Pokyny pro vypracování

1) Seznamte se s různými formáty pro ukládání řídkých matic (COO, CSR, CSC).
2) Nastudujte tématiku LU rozkladu pro řídké matice [1-3], seznamte se předchozími pracemi na toto téma, zejména [4-6].
3) Seznamte se s heuristikami, které byly implementovány v [4-6] a které snižují počet nově vzniklých prvků (tzv. fill-in), minimálně s těmito: Multiple Minimum Degres ordering, Reverse Cuthill-Mckee, Markowitz strategy.
4) Naimplementujte LU rozklad pro řídké matice pomocí vícevláknového Doolittlova algoritmu.
5) Pro heuristiky z bodu 3) diskutujte možnosti jejich vícevláknové paralelizace pod sdílenou pamětí pomocí technologie OpenMP.
6) Výsledné implementace otestujte na řídkých maticích získaných z veřejně dostupných zdrojů (MatrixMarket, ...) a porovnejte výslednou časovou a paměťovou náročnost pro testované matice.

Seznam odborné literatury

[1] THOMPSON, Erik G.: Doolittle Decomposition of a Matrix [online]. 2005, [cit. 2016-03-17].
Dostupné z: http://www.engr.colostate.edu/ thompson/hPage/CourseMat/Tutorials/CompMethods/doolittle.pdf
[2] HEGGERNES, P., S.C. EISENSTAT, G. KUMFERT, a A. POTHEN: The Computational Complexity of the Minimum Degree Algorithm. 2001,[cit. 2016-03-19].
Dostupné z: http://oai.dtic.mil/oai/oai?verb=getRecord&metadataPrefix=html&identifier=ADA398632
[3] LI, Xiaoye S., James W. DEMMEL, John R. GILBERT, Laura GRIGORI, Meiyue SHAO a Ichitaro YAMAZAKI: SuperLU Users' Guide. 2011, [cit. 2016-05-11].
Dostupné z: http://crd-legacy.lbl.gov/ xiaoye/SuperLU/superlu_ug.pdf
[4] TURČAN, Lukáš: Řešiče rozsáhlých soustav lineárních rovnic. BP, ČVUT FIT, 2015.
[5] KUSÝ, Stanislav: Efektivní LU rozklad pro řídké matice. BP, ČVUT FIT, 2015.
[6] Turcajová, Gabriela: Efektivní LU rozklad pro řídké matice, BP, ČVUT FIT 2016.

doc. Ing. Jan Janoušek, Ph.D.
vedoucí katedry

prof. Ing. Pavel Tvrdík, CSc.
děkan

V Praze dne 9. února 2017

České vysoké učení technické v Praze
Fakulta informačních technologií
Teoretická Informatika

Bakalářská práce

Efektivní LU rozklad pro řídké matice

Lukáš Jurásek

Vedoucí práce: doc. Ing. Ivan Šimecek, Ph.D.

16. května 2017

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 16. května 2017 .....

Odkaz na tuto práci

Jurásek, Lukáš. *Efektivní LU rozklad pro řídké matice.* Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2017.

Abstrakt

Tato bakalářská práce se zabývá implementací a analýzou algoritmů využívaných pro LU faktorizaci řídkých matic. Literární rešerše obsahuje teoretické základy potřebné pro pochopení algoritmů, kterými se práce zabývá. Tyto algoritmy jsou zde také popsány a vysvětleny. Praktická část obsahuje popis konkrétní implementace a její analýzu. Implementace je realizována v jazyku C++ za pomoci knihovny OpenMP, která je použita pro pararelizaci. Hlavním výsledkem této práce je sada dat vypovídajících o vlastnostech a schopnostech zvolených algoritmů. Tyto údaje mohou pomoci zájemcům při volbě algoritmů pro použití v aplikacích využívající řídké matice. V příloze práce lze nalézt zdrojové kódy implementace testované v této práci.

Klíčová slova LU faktorizace, řídké matice, implementace, analýza, heuristika

Abstract

This bachelor thesis deals with the implementation and analysis of algorithms used for LU factorization of sparse matrices. The literary research contains the theoretical foundations needed to understand the algorithms that the thesis

deals with. These algorithms are also described and explained here. The practical part contains a description of the selected implementation and its analysis. Implementation is implemented in C++ using the OpenMP library, which is used for parallelization. The main result of this work is a set of data showing the properties and abilities of the chosen algorithms. This data can help anyone interested in selecting algorithms for use in applications using sparce matrices. In the appendix, the implementation source codes tested in this work can be found.

Keywords LU factorization, sparce matrices, implementation, analysis, heuristic

Obsah

Seznam obrázků

Úvod

V dnešní době se staly počítače a Internet součástí každodenního života pro velkou část populace. Velké množství aplikací, jako například doporučovací systémy, strojové učení a počítačová grafika, využívá pro reprezentaci zpracovávaných dat právě řídké matice. Na značné množství těchto aplikací je kladen důraz na jejich rychlost, a zde se nachází užitek této práce. Výsledky této práce napomohou při tvorbě aplikací využívající řídké matice.

Zároveň také poslouží jako jako naučný materiál pro zájemce o téma řídkých matic či efektivního programování.

Práce se zabývá implementací a analýzou Doolittlova algoritmu pro LU faktorizaci řídkých matic a heuristik snažících se minimalizovat tak zvaný „fill-in“. 1.část práce se věnuje seznámením s algoritmy, heuristikami a nezbytnými teoretickými základy. 2.část se věnuje implementaci Doolittlova algoritmu a heuristik. 3.část se věnuje analýze výsledné implementace Doolittlova algoritmu a heuristik.

Kapitola 1

Teoretický základ

LU faktorizace se v praxi používá především ke zrychlení řešení lineárních rovnic s více pravými stranami.

Jelikož tato práce se zabývá maticemi a algoritmy s nimi pracujícími, nachází se v této kapitole definice matice a s maticemi související pojmy využité v této práci. Zároveň je zde seznámení se s různými formáty uložení matic v programech.

Některé heuristiky obsažené v této práci také potřebují k práci grafy, proto tato kapitola obsahuje také definice grafu a pojmů s grafy souvisejícími.

1.1 Matice

V této sekci je čerpáno z [1].

Matice Uspořádaný soubor mn čísel, kde $m, n \in \mathbb{N}$, uspořádaných do tabulky o m řádcích a n sloupcích nazýváme **matice** $m \times n$. Značíme například takto:

$$\mathbb{A}_{m,n} = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{pmatrix}$$

kde $a_{i,j}$ je prvek matice na i-tém řádku a j-tém sloupci. Číslo i nazýváme řádkový index a j sloupcový index. Množinu všech matic typu $m \times n$ značíme $\mathbb{R}^{m,n}$.

Čtvercová matice Čtvercová matice je speciální případ obecné matice, kde $m = n$. Nazýváme také **matice** $n \times n$. Množinu všech matic typu $n \times n$ značíme $\mathbb{T}^{n,n}$.

Vektor Vektorem nazýváme uspořádanou n-tici prvků, kde $n \in \mathbb{N}$. Značíme $\mathbb{A} = (a_1, a_2, \ldots, a_n)$.

Diagonála matice Diagonálou čtvercové matice $\mathbb{A} \in \mathbb{T}^{n,n}$ je vektor

$$(a_{1,1}, a_{2,2}, \ldots, a_{n,n}) \in \mathbb{T}^n$$

.

Horní/Dolní trojúhelníková matice Horní trojúhelníková matice je matice $\mathbb{A} \in \mathbb{R}^{m,n}$, pro kterou platí $\forall i > j : a_{i,j} = 0$. Analogicky pro dolní trojúhelníkovou matici platí $\forall i < j : a_{i,j} = 0$.

Násobení matic Pokud máme matice $\mathbb{A} \in \mathbb{R}^{m,n}$ a $\mathbb{B} \in \mathbb{R}^{n,p}$, kde $m, n, p \in \mathbb{N}$, tak jejich součinem bude matice $\mathbb{C}$ a $\mathbb{B}$ je matice $\mathbb{C} \in \mathbb{R}^{m,p}$ pro kterou platí:

$$c_{i,j} = \sum_{k=1}^{n} a_{i,k} b_{k,j}$$

Značíme $\mathbb{AB} = \mathbb{C}$.

Jednotková matice Jednotková matice je matice $\mathbb{E} \in \mathbb{T}^{n,n}$ pro kterou platí

$$\mathbb{E}_{ij} = \delta_{ij} \quad i, j \in \hat{n}$$

Pro δ_{ij} platí $\delta_{ij} = \begin{cases} 1, & \text{pro } i = j \\ 0, & \text{jinak} \end{cases}$.

Regulární a singulární matice Mějme matici $\mathbb{A} \in \mathbb{T}^{n,n}$. Existuje-li matice $\mathbb{B} \in \mathbb{T}^{n,n}$ taková, že platí

$$\mathbb{AB} = \mathbb{BA} = \mathbb{E}$$

nazýváme matici $\mathbb{A}$ **regulární** a matici $\mathbb{B}$ **inverzní** k $\mathbb{A}$. Značíme $\mathbb{B} = \mathbb{A}^{-1}$. Pokud $\mathbb{A}$ není regulární, nazýváme ji **singulární**.

Transponovaná matice Transpozicí matice $\mathbb{A} \in \mathbb{R}^{m,n}$ je matice z $\mathbb{R}^{n,m}$ pro kterou platí, že její prvek v j-tém řádku a i-tém sloupci se rovná $a_{i,j}$. Tuto matici značíme $\mathbb{A}^T$.

Symetrická matice Matici nazveme symetrickou, když pro ni platí $\mathbb{A} = \mathbb{A}^T$.

LU rozklad *LU rozkladem* regulární matice $\mathbb{A}$ je rozklad na dolní trojúhelníkovou matici $\mathbb{L}$ a horní trojúhelníkovou matici $\mathbb{U}$. Pro tyto matice platí $\mathbb{LU} = \mathbb{A}$.

Řídká matice Řídká matice nemá přesnou definici. Obecně pro řídké matice platí, že většina jejich prvků je nulových. Tato hranice však není nikde přesně stanovena. Analogicky, pokud je většina prvků matice nenulových, nazveme ji maticí hustou.

1.2 Formáty uložení řídkých matic

V této sekci je čerpáno z [2] a [3].

1.2.1 Hustý formát

Hustý formát uložení matice je formát bez jakékoli komprese. Matice uložené v tomto formátu jsou uloženy jako dvourozměrné pole hodnot, kde hodnota matice $a_{i,j}$ je uložena v poli na adrese $A[i, j]$. Dvourozměrná pole jsou v programovacích jazycích většinou uložena jako pole polí, což umožňuje jednoduché a rychlé procházení a hledání hodnot v matici.

Pro řídké matice, zvláštně rozsáhlé, ale tento formát není příliš ideální. Obsahuje velké množství redundantních dat v podobě nulových prvků, které v případě rozsáhlých matic zabírají značné množství paměti. Pro malé řídké matice však tento formát může být atraktivní díky své rychlosti, ale musí se počítat s větším využitím paměti.

1.2.2 COO formát

Souřadnicový formát (Coordinate format) používá pro ukládání nenulových prvků matice 3 jednorozměrná pole s velikostí stejnou jako je počet nenulových prvků v matici. Tyto pole jsou *Row, Column* a *Data*, tedy řádek, sloupec a data. V ideálním případě by tyto pole byla seřazena primárně podle čísla řádku a sekundárně podle čísla sloupce pro rychlejší vyhledávání, ale toto uspořádání není explicitně vyžadováno.

Tento formát je více paměťově náročný oproti jiným komprimovaným formátům uložení, je ale vhodný pro postupnou konstrukci matice a převod do jiných formátu uložení.

Ukázka uložení matice ve formátu COO

$$\mathbb{A} = \begin{pmatrix} 4 & 0 & 0 & 6 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 9 & 0 & 0 & 0 & 3 \\ 0 & 0 & 0 & 7 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 8 \end{pmatrix}$$

Matice $\mathbb{A}$ ve formátu CCO při indexování od 1:

Row	1	1	2	4	4	4	5	6
Column	1	4	2	1	2	6	4	6
Data	4	6	3	1	9	3	7	8

1.2.3 CSR formát

Compressed Sparse Row formát používá pro ukládání nenulových prvků 3 jednorozměrná pole, z nichž 2 pole mají velikost stejnou jako je počet nenulových prvků, a třetí je velké jako je počet řádků matice plus 1. Tyto pole jsou *Index, Pointer* a *Data*. Pole *Data* obsahuje nenulové prvky matice v pořadí primárně podle čísla řádku a sekundárně podle čísla sloupce. Pole *Index* obsahuje sloupcové indexy korespondující k datům z pole *Data* v identickém pořadí. Pole *Pointer* obsahuje data definována touto rekurzivní funkcí: $\text{Pointer}\,[0] = 0 : \quad \text{Pointer}\,[i] = \text{Pointer}\,[i-1] + (nnz_{i-1})$, kde nnz_{i-1} je počet nenulových prvků na řádku $(i-1)$ původní matice. Prvních m hodnot pole Pointer tedy obsahuje indexy prvního nenulového prvku daného řádku v polích Index a Data a prvek $m+1$ obsahuje celkový počet nenulových prvků v matici. Je zajímavé podotknout, že jelikož první prvek pole Pointer je vždy 0 a poslední je vždy počet nenulových prvků v matici, jsou tyto hodnoty v podstatě redundantní. Je ale výhodné tyto hodnoty v poli ponechat, jelikož díky těmto hodnotám není potřeba zvlášť ošetřovat mezní případy přístupu do takto uložené matice. Tyto hodnoty tak zajišťují platnost $\text{Pointer}\,[i+1] - \text{Pointer}\,[i] =$ (počet nenulových prvků na i-tém řádku).

Tento formát je vhodný pokud je potřeba procházet matici po řádcích.

Ukázka uložení matice ve formátu CSR

$$\mathbb{A} = \begin{pmatrix} 4 & 0 & 0 & 6 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 9 & 0 & 0 & 0 & 3 \\ 0 & 0 & 0 & 7 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 8 \end{pmatrix}$$

Matice $\mathbb{A}$ ve formátu CRS při indexování od 1:

Index	1	4	2	1	2	6	4	6
Data	4	6	3	1	9	3	7	8
Pointer	0	2	3	3	6	7	8	

1.2.4 CSC formát

Compressed Scarse Column formát je velmi podobný formátu CRS, rozdíl mezi nimi je že CSR ukládá data po řádcích, CSC ukládá data po sloupcích. Využívá k uložení matice 3 jednorozměrná pole, stejně jako CSR. Namísto

počtu nenulových prvků v řádku se ve vzorci pro výpočet hodnot v poli *Pointer* využívá počet nenulových prvků ve sloupci a v poli *Index* jsou namísto indexů sloupců indexy řádků.

Tento formát je vhodný pokud je potřeba procházet matici po sloupcích.

Ukázka uložení matice ve formátu CSC

$$\mathbb{A} = \begin{pmatrix} 4 & 0 & 0 & 6 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 9 & 0 & 0 & 0 & 3 \\ 0 & 0 & 0 & 7 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 8 \end{pmatrix}$$

Matice $\mathbb{A}$ ve formátu CRS při indexování od 1:

Index	1	4	2	4	1	5	4	6
Data	4	1	3	9	6	7	3	8
Pointer	0	2	4	4	6	6	8	

1.2.5 Matrix Market

Formát Matrix Market je využívaný stejnojmenným veřejným zdrojem matic. Tento formát má 2 druhy: „Coordinate“ využívaný k ukládání řídkých matic a „Array“ využívaný pro ukládání hustých matic. Tento formát se využívá pro ukládání matic do souborů na rozdíl od formátu zmíněných v předchozích částech práce.

Jelikož se tato práce zabývá řídkými maticemi, je zde vysvětlen pouze *Coordinate* formát. Data v souboru mají ve formátu Matrix Market pevně danou strukturu. Na prvním řádku se nachází hlavička. Ta povinně začíná řetězcem „%%MatrixMarket“. Následují informace o matici: formát uložení, datový typ prvků a typ matice. Typ matice je buďto „general“ který říká, že v souboru budou všechny nenulové prvky, anebo „symmetric“ který říká, že soubor obsahuje pouze nenulové prvky na diagonále a přímo pod diagonálou. V tomto případě jsou totiž prvky nad diagonálou redundantní, jsou totiž totožné s prvky pod diagonálou.

Na dalších řádcích může být dobrovolný komentář. Počet řádků komentáře není omezen, všechny řádky komentáře však musí začínat znakem „%“.

Řádek následující ihned po komentáři obsahuje tři čísla v tomto pořadí: počet řádků matice, počet sloupců matice a počet nenulových prvků matice.

Zbylé řádky souboru obsahují nenulové prvky matice. Ty jsou v následujícím formátu: index řádku, index sloupce a hodnota.

Ukázka matice uložené ve formátu Matrix Market

$$\mathbb{A} = \begin{pmatrix} 4 & 0 & 0 & 6 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 9 & 0 & 0 & 0 & 3 \\ 0 & 0 & 0 & 7 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 8 \end{pmatrix}$$

```
%%MatrixMarket matrix coordinate integer general
%
% Dobrovolný komentář
%
6 6 8
1 1 4
1 4 6
2 2 3
4 1 1
4 2 9
4 6 3
5 4 7
6 6 8
```

1.3 Grafy

V této sekci je čerpáno z [4].

Neorientovaný graf Neorientovaný graf je uspořádaná dvojice (V, E), kde

- V je neprázdná konečná množina vrcholů (nebo také uzlů)
- E je množina hran

Hrana je neuspořádaná dvojice vrcholů, platí tedy $E \subseteq 2^V$. Množinu všech dvouprvkových podmnožin množiny V označujeme $\binom{V}{2}$.

Prostý graf Graf nazýváme prostý v případě, že žádné 2 uzly nejsou spojeny více jak jednou hranou, tedy neobsahuje rovnoběžné hrany.

Smyčka grafu Smyčka grafu je speciální případ hrany, která spojuje uzel v grafu sám se sebou.

Obyčejný graf Obyčejný graf je takový graf, který neobsahuje smyčky.

Úplný graf Úplný graf na n vrcholech je graf $\left(V, \binom{V}{2}\right)$, kde $|V| = n$.

Podgraf Graf H je podgrafem grafu G, pokud platí $V(H) \subseteq V(G)$ a zároveň $E(H) \subseteq E(G)$. Toto značíme $H \subseteq G$.

Stupeň uzlu Nechť máme graf $G = (V, E)$ a vrchol $v \in V$. Stupněm uzlu v je počet hran grafu G, které obsahují uzel v. Značíme jako $deg_G(v)$.

Klika grafu Klikou nazveme úplný podgraf grafu G.

Matice sousednosti Nechť máme graf $G = (V, E)$, kde $V = (v_1, v_2, \ldots, v_n)$. Maticí sousednosti nazveme čtvercovou matici $A_G = (a_{ij})_{i,j=1}^{n}$ pro kterou platí předpis $a_{i,j} = \begin{cases} 1, & \text{když } \{v_i, v_j\} \in E \\ 0, & \text{jinak} \end{cases}$.

Kapitola 2

Doolittlův algoritmus pro LU faktorizaci

Doolittlův algoritmus[5] pro LU faktorizaci je založený na Gaussově eliminaci. Mezi jeho výhody patří jeho jednoduchost na pochopení a implementaci, a to i pro složitější formáty uložení řídkých matic. Algoritmus je také zároveň důkazem, že daná faktorizace je unikátní. Je také možné algoritmus implementovat „in-place“, tedy s přidanou pamětí konstantní $\mathcal{O}(1)$ velikosti. Tento přístup má však nevýhodu ztráty původní matice, jelikož jsou na její místo ukládány hodnoty z obou výsledných matic.

Algoritmus provede LU faktorizaci matice $\mathbb{A}$ na dolní trojúhelníkovou matici $\mathbb{L}$ a horní trojúhelníkovou matici $\mathbb{U}$. Pro prvky matice $\mathbb{L}$ platí, že prvky na její diagonále jsou rovné 1 a prvky pod diagonálou jsou určeny

$$l_{i,j} = \frac{a_{i,j} - \sum_{k=1}^{j-1} l_{i,k} u_{k,j}}{u_{j,j}}$$

a pro matici $\mathbb{U}$ platí, že její prvky na a zároveň nad diagonálou jsou určeny

$$u_{i,j} = a_{i,j} - \sum_{k=1}^{i-1} l_{i,k} u_{k,j}.$$

Z těchto vztahů je vidět, že po přečtení hodnoty $a_{i,j}$ už není nikdy využita. Toto a zároveň fakt, že prvky na diagonále matice $\mathbb{L}$ jsou rovné 1, tedy v podstatě redundantní, znamená, že prvky jak matice $\mathbb{L}$ i matice $\mathbb{U}$ lze uložit zpět do matice $\mathbb{A}$, a tím odstranit nutnost alokovat dodatečné místo v paměti na uložení matic $\mathbb{L}$ a $\mathbb{U}$. To znamená, že algoritmus je možné implementovat „in-place“, jak bylo zmíněno v předchozím odstavci.

2.1 Průběh algoritmu

I když je slovní popis algoritmu postačující pro jeho definici, pro jeho vysvětlení a pochopení je mnohem užitečnější názorný příklad s vysvětlením jednotlivých kroků. Z tohoto důvodu následuje názorný příklad na matici o rozměrech 3×3.

$$\begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{pmatrix} = \begin{pmatrix} l_{1,1} & & \\ l_{2,1} & l_{2,2} & \\ l_{3,1} & l_{3,2} & l_{3,3} \end{pmatrix} \begin{pmatrix} u_{1,1} & u_{1,2} & u_{1,3} \\ & u_{2,2} & u_{2,3} \\ & & u_{3,3} \end{pmatrix}$$

$$\begin{pmatrix} 2 & -1 & -2 \\ -4 & 6 & 3 \\ -4 & -2 & 8 \end{pmatrix} = \begin{pmatrix} 1 & & \\ . & 1 & \\ . & . & 1 \end{pmatrix} \begin{pmatrix} . & . & . \\ & . & . \\ & & . \end{pmatrix}$$

Tento zápis reprezentuje matice využité v Doolittlově algoritmu před jeho spuštěním. Tečky označují prvky k nalezení.

Algoritmus začíná vyplněním prvního řádku v matici $\mathbb{U}$.

$$\begin{pmatrix} 2 & -1 & -2 \\ -4 & 6 & 3 \\ -4 & -2 & 8 \end{pmatrix} = \begin{pmatrix} 1 & & \\ . & 1 & \\ . & . & 1 \end{pmatrix} \begin{pmatrix} 2 & -1 & -2 \\ & . & . \\ & & . \end{pmatrix}$$

Tyto prvky jsou vypočteny dle vzorce uvedeném v definici algoritmu.

Následuje vyplnění prvního sloupce matice $\mathbb{L}$.

$$\begin{pmatrix} 2 & -1 & -2 \\ -4 & 6 & 3 \\ -4 & -2 & 8 \end{pmatrix} = \begin{pmatrix} 1 & & \\ -2 & 1 & \\ -2 & . & 1 \end{pmatrix} \begin{pmatrix} 2 & -1 & -2 \\ & . & . \\ & & . \end{pmatrix}$$

Podobně jako v předchozím kroku, nové prvky jsou vypočteny dle vzorce uvedeném v definici algoritmu.

Zbylé kroky jsou identické jako oba předchozí kroky, popisky už tedy nejsou potřeba.

$$\begin{pmatrix} 2 & -1 & -2 \\ -4 & 6 & 3 \\ -4 & -2 & 8 \end{pmatrix} = \begin{pmatrix} 1 & & \\ -2 & 1 & \\ -2 & . & 1 \end{pmatrix} \begin{pmatrix} 2 & -1 & -2 \\ & 4 & -1 \\ & & . \end{pmatrix}$$

$$\begin{pmatrix} 2 & -1 & -2 \\ -4 & 6 & 3 \\ -4 & -2 & 8 \end{pmatrix} = \begin{pmatrix} 1 & & \\ -2 & 1 & \\ -2 & -1 & 1 \end{pmatrix} \begin{pmatrix} 2 & -1 & -2 \\ & 4 & -1 \\ & & . \end{pmatrix}$$

$$\begin{pmatrix} 2 & -1 & -2 \\ -4 & 6 & 3 \\ -4 & -2 & 8 \end{pmatrix} = \begin{pmatrix} 1 & & \\ -2 & 1 & \\ -2 & -1 & 1 \end{pmatrix} \begin{pmatrix} 2 & -1 & -2 \\ & 4 & -1 \\ & & 3 \end{pmatrix}$$

Jak již bylo zmíněno, algoritmus lze implementovat „in-place“, tedy obě výsledné matice uložit do matice původní. Pro tento příklad by výsledná matice vypadala takto:

$$\begin{pmatrix} 2 & -1 & -2 \\ -2 & 4 & -1 \\ -2 & -1 & 3 \end{pmatrix}.$$

2.2 Pseudokód algoritmu

Algorithm 1 Doolittlův algoritmus

Vstup: Matice $\mathbb{A}(dim, dim)$
Výstup: Matice $\mathbb{L}(dim, dim), \mathbb{U}(dim, dim)$

```
1:  function DOOLITTLE
2:      for i ← 1 to dim do
3:          for j ← i to dim do
4:              u(i, j) ← a(i, j)
5:              for k ← 1 to i − 1 do
6:                  u(i, j) ← u(i, j) − l(i, k) ∗ u(k, j)
7:              end for
8:          end for
9:          for j ← i + 1 to dim do
10:             l(i, j) ← a(i, j)
11:             for k ← 1 to i − 1 do
12:                 l(i, j) ← l(i, j) − l(j, k) ∗ u(k, i)
13:             end for
14:             l(i, j) ← l(i, j) /u(i, i)
15:         end for
16:     end for
17: end function
```

2.3 Složitost algoritmu

2.3.1 Časová složitost

Podle pseudokódu lze vyvodit časová složitost

$$\mathcal{O}\left(\sum_{i=1}^{n}\left(\sum_{j=i}^{n}\sum_{k=1}^{i-1}1+\sum_{j=i+1}^{n}\sum_{k=1}^{i-1}1\right)\right)\approx\mathcal{O}\left(n^3\right).$$

Časová složitost algoritmu je tedy kubická a záleží na dimenzi vstupní matice. Tato složitost ovšem platí pouze pro formáty uložení s konstantním časem

přístupu a zápisu, tedy hlavně hustý formát uložení. Pro komprimované formáty uložení řídkých matic je složitost vyšší, a to o složitost čtení a zápisu do matice.

2.3.2 Paměťová složitost

Přesná paměťová složitost algoritmu záleží na způsobu implementace a formátu uložení matic. Pro implementaci popsanou pseudokódem v této práci a hustý formát uložení matic je složitost $\mathcal{O}(2n^2)$. Toto můžeme snížit na $\mathcal{O}(n^2)$ pokud obě výsledné matice $\mathbb{L}$ a $\mathbb{U}$ uložíme do jedné matice. Avšak pokud použijeme „in-place“ implementaci algoritmu, můžeme složitost snížit na $\mathcal{O}(1)$. Tento přístup ale není vždy hodný, jelikož tento způsob implementace přepíše vstupní data.

Situace je složitější pokud je použit některý z formátu uložení pro řídké matice. Nadále bude uvažován formát uložení CSR nebo CSC. Možnosti implementace jsou stejné jako pro hustý formát uložení, avšak paměťové složitosti se liší. Při uložení výsledků do nových matic, je potřeba $\mathcal{O}(2nnz_L + 2nnz_U + 2n)$ paměti navíc, kde nnz je počet nenulových prvků v matici. Tento přístup má výhodu volnosti ve volbě formátu uložení pro obě výsledné matice. Můžeme také obě výsledné matice uložit to jedné matice. Při tomto přístupu ušetříme oproti předchozímu případu $\mathcal{O}(n)$ paměti, avšak ztratíme volbu výběru formátu uložení pro obě matice nezávisle. Lze také uložit výsledky do vstupní matice. V tomto případě je paměťová složitost $\mathcal{O}(nnz_L + nnz_U - nnz_A)$, ztratíme však volbu formátu uložení výstupních matic a zároveň ztratíme vstupní data.

2.4 Možnosti paralelizace

Možnosti paralelizace toho algoritmu jsou limitované, ale existují. Z pseudokódu algoritmu lze vyčíst nemožnost paralelizace hlavního cyklu, jelikož jeho $i+1$-ní iterace přímo závisí na výsledku i-té iterace. Vnitřní cykly j a k už paralelizovat jdou.

V cyklu j jsou výsledky iterací na sobě nezávislé, lze tedy snadno paralelizovat rozdělením iterací cyklu mezi dostupná vlákna. Situaci stěžuje využití formátu CSR a CSC, který znemožňuje nezávislé přidávání nových prvků do matice. Toto lze vyřešit, avšak je nastane zpomalení kvůli nutnosti ošetření přístupu do matice.

Cyklus k lze také snadno paralelizovat. Existuje sice hazard v podobě změny prvků $l(i,j)$ a $u(i,j)$, toto však lze snadno vyřešit pomocí mechanizmu *reduction* dostupným v knihovně OpenMP.

Kapitola 3

Heuristiky pro snížení počtu nově vzniklých nenulových prvků

Při operacích s řídkými maticemi je často kvůli jejich rozměrům požadováno zachování řídkosti ve výsledných maticích. Existuje celá řada heuristik snažící se minimalizovat tzv. „fill-in“, tedy vytvoření nových nenulových prvků ve výsledných maticích. V této kapitole jsou popsány některé z nich.

3.1 Symetrické matice

Následující algoritmy pro fungování potřebují možnost vstupní matici interpretovat jako matici sousednosti. Je tedy požadována její symetričnost.

Tyto algoritmy však lze použít na nesymetrické matice, a to když se k nesymetrické vstupní matici přičte matice transponovaná, tedy $\mathbb{A}^S = \mathbb{A} + \mathbb{A}^T$.

3.1.1 Minimum Degree Ordering

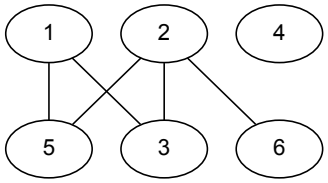
Hlavní ideou algoritmu MD[6] je minimalizace fill-in odstraňováním uzlů z eliminačního grafu v pořadí podle jejich stupně aktuálního stupně. Eliminační graf se získá vytvořením grafu matice sousednosti. Jako matice sousednosti se bere samotná vstupní matice s ignorovanými smyčkami, tedy prvky na diagonále. Výstupem algoritmu je permutační vektor obsahující indexy uzlů v pořadí v jakém byli smazány z eliminačního grafu.

Algoritmus začne nalezením uzlu v eliminačním grafu s nejnižším stupněm. Jeho index uloží na počátek vektoru, který na konci algoritmu bude obsahovat permutační vektor. Tento uzel z grafu vymaže a vytvoří kliku obsahující sousedy smazaného uzlu. Vytvoření kliky dosáhne sjednocením vlastním seznamem sousedů se seznamy sousedů svých sousedů. Při sjednocování je třeba nepřidat do seznamu sousedů uzlu sám onen uzel, aby se zabránilo vzniku smyček. Ze seznamu sousedů se také smaže smazaný uzel. Toto se opakuje dokud eliminační graf obsahuje uzly ke smazání.

3.1.1.1 Průběh algoritmu

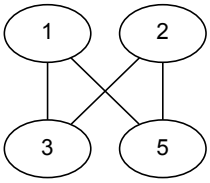
$$\mathbb{A} = \begin{pmatrix} 1 & 0 & 4 & 0 & 3 & 0 \\ 0 & 3 & 0 & 0 & 3 & 8 \\ 6 & 0 & 5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 8 & 0 & 0 \\ 2 & 6 & 0 & 0 & -1 & 0 \\ 0 & 9 & 0 & 0 & 0 & 6 \end{pmatrix}$$

Tuto matici si algoritmus převede na následující graf.



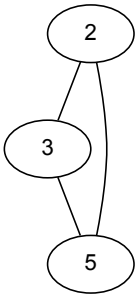
Obrázek 3.1: Počáteční graf MDO

Jelikož má uzel 4 nejnižší stupeň, je z grafu odstraněn a přidán do permutačního vektoru. Jako další je smazán uzel 6. Je přidán na konec permutačního vektoru.



Obrázek 3.2: Graf MDO po odstranění uzlů 4 a 6

V aktuálním grafu mají všechny uzly stejný. V tomto případě je vybrán uzel 1, jelikož je to uzel s nejnižším indexem.



Obrázek 3.3: Graf MDO po odstranění uzlu 1

Takto algoritmus pokračuje dokud nejsou všechny uzly smazány. Na konci běhu algoritmu je výsledný vektor | 4 | 6 | 1 | 2 | 3 | 5 | .

3.1.1.2 Pseudokód algoritmu

Algorithm 2 Minimum Degree Ordering

Vstup: Matice $\mathbb{A}(dim, dim)$
Výstup: Permutační vektor *perm*

```
1: function MDO
2:     while perm.size < dim do
3:         node ← graph.min
4:         perm.pushback (node)
5:         for i ← 1 to node.neigh.size do
6:             node.neigh.delete (node)
7:             node.neigh.union (node.neigh)
8:         end for
9:     end while
10: end function
```

3.1.2 Multiple Minimum Degree Ordering

Multiple Minimum Degree Ordering je vylepšení algoritmu Minimum Degree Ordering navrhnuto H. Liu. Dle jeho pozorování, pokud eliminační graf obsahuje více uzlů se stejným minimálním stupněm, tak po smazání jednoho z nich určitě budou zbylé uzly mezi těmi s minimálním stupněm. Tato modifikace tedy odstraňuje všechny uzly se stejným minimálním stupněm v jednom kroku. Motivací za touto modifikací je snaha snížit čas potřebný k přepočtu stupňů uzlů afektovaných při mazání uzlů z eliminačního grafu.

V příkladu běhu algoritmu budou odstraněny uzly $1, 2, 3$ a 5 v jednom kroku. Pořadí přidání uzlů do permutačního vektoru není explicitně zadáno.

3.1.2.1 Možnosti paralelizace

Pararelizace algoritmu Multiple Minimum Degree Ordering je poměrně limitovaná. Nejlépe lze paralelizovat vytváření klik v grafu, jelikož můžeme zpracovávat několik sousedních uzlů smazaného uzlu, jelikož tyto operace jsou na sobě nezávislé. U MMDO by se také mohlo zdát že zpracovávání několika smazaných uzlů najednou je výhodné, tento přístup však přináší značné množství hazardů a vyžaduje tedy poměrně agresivní synchronizaci.

3.1.3 Cuthill-McKee

Algoritmus Cuthill-McKee[7] se snaží minimalizovat „fill-in“ průchodem grafu vzniklého z matice sousednosti ve směru nejnižšího stupně. Jako matice sou-

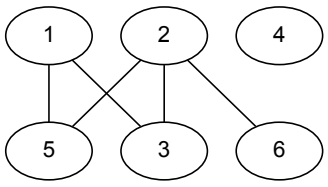
sednosti se bere vstupní matice se zanedbanými smyčkami, stejně jako v algoritmu MD. Algoritmus je podobný algoritmu Breath-first search se změněným pořadím procházením uzlů. Výstupen algoritmu je permutační vektor.

Algoritmus začne selekcí uzlu z grafu s nejnižším stupněm. Ten je vložen do permutačního vektoru. Do předpřipravené fronty jsou vloženy sousedé vybraného stupně v pořadí dle jejich stupně. Následně je vybrán uzel z fronty a je pro něj opakován stejný proces jako pro první uzel. Toto se opakuje, dokud fronta není prázdná. Pokud se fronta vyprázdní a délka permutačního vektoru se nerovná dimenzi vstupní matice, vybere se nový, zatím nevybraný uzel s nejnižším stupněm a algoritmus pokračuje.

3.1.3.1 Průběh algoritmu

$$\mathbb{A} = \begin{pmatrix} 1 & 0 & 4 & 0 & 3 & 0 \\ 0 & 3 & 0 & 0 & 3 & 8 \\ 6 & 0 & 5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 8 & 0 & 0 \\ 2 & 6 & 0 & 0 & -1 & 0 \\ 0 & 9 & 0 & 0 & 0 & 6 \end{pmatrix}$$

Tato matice je převedena na tento graf.



Obrázek 3.4: Graf pro Cuthill-McKee

Ten je procházen podle definice algoritmu. Výsledný permutační vektor je

4	6	2	3	1	5

. Pro případ že má algoritmus více možných cest, není zadefinováno pravidlo pro rozhodnutí. V tomto případě měly prioritu uzly s nižším indexem.

3.1.3.2 Pseudokód algoritmu

Algorithm 3 Cuthill-McKee

Vstup: Matice $\mathbb{A}(dim, dim)$
Výstup: Permutační vektor *perm*

```
1:  function CM
2:      while perm.size < dim do
3:          node ← graph.min
4:          perm.pushback (node)
5:          queue.push (node.neigh)
6:          while !queue.empty do
7:              node ← queue.pop
8:              perm.pushback (node)
9:              queue.push (node.neigh)
10:         end while
11:     end while
12: end function
```

3.1.4 Reverse Cuthill-McKee

Reverse Cuthill-McKee je modifikace algoritmu Cuthill-McKee Alanem Georgem. Modifikace spočívá v obrácení výsledného permutačního vektoru.

3.1.4.1 Možnosti paralelizace

Možnosti paralelizace tohoto algoritmu jsou velmi limitované. Jednoduchá a efektivní část k paralelizaci je obracení permutačního vektoru. Ve zbytku algoritmu je každý krok závislý na kroku předešlém, není tedy možno rozdělení algoritmu na části pro paralelizaci. Rozdělení by způsobilo značnou změnu výsledku.

3.2 Nesymetrické matice

Následující heuristika je určena pro nesymetrické matice. Nesymetričnost však není podmínkou, lze tak použít i na symetrické matice.

3.2.1 Markowitzova strategie

Markowitzova strategie[8] se snaží minimalizovat „fill-in“ pomocí permutace řádků a sloupců matice na základě tzv. Markowitzových cen.

Algoritmus v každé svojí iteraci pracuje s pravou dolní částí matice o velikosti $(n-i) \times (n-i)$, kde $i \in \langle 0, n-2 \rangle$ je aktuální iterace algoritmu. Pro každý nenulový prvek aktuální matice je spočtena tzv. Markowitzova cena

definována

$$price\left(a_{j,k}\right) = \left(nnz_r\left(j\right) - 1\right) * \left(nnz_c\left(k\right) - 1\right)$$

kde $nnz_r\left(j\right)$ je počet nenulových prvků j-tého řádku aktuální matice a $nnz_c\left(k\right)$ počet nenulových prvků k-tého sloupce aktuální matice. Po vypočtení těchto cen jen zvolen řádek a sloupec obsahující prvek s nejnižší Markowitzovou cennou. Zvolený řádek se přesune na první řádek v aktuální matici, sloupec na první sloupec v aktuální matici. Toto se opakuje pro všechny hodnoty $i \in \langle 0, n-2 \rangle$.

3.2.1.1 Průběh algoritmu

$$\mathbb{A} = \begin{pmatrix} 1 & 0 & 4 & 0 & 3 & 0 \\ 0 & 3 & 0 & 0 & 3 & 8 \\ 6 & 0 & 5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 8 & 0 & 0 \\ 2 & 6 & 0 & 0 & -1 & 0 \\ 0 & 9 & 0 & 0 & 0 & 6 \end{pmatrix}$$

Pro tuto matici algoritmus spočítá Markowitzovy ceny jednotlivých nenulových prvků

$$\begin{pmatrix} 4 & & 2 & & 4 & \\ & 4 & & & 4 & 2 \\ 2 & 0 & 1 & & & \\ & & & 0 & & \\ 4 & 4 & & & 4 & \\ & 2 & & & & 1 \end{pmatrix}$$

Prvek s minimální Markowitzovo cennou je $a_{4,4}$, je tedy prohozen řádek 0 a 4 a sloupec 0 a 4. V Dalším kroku se pracuje s pravým dolním rohem matice o rozměru 5×5.

$$\bar{\mathbb{A}} = \begin{pmatrix} 3 & 0 & 0 & 3 & 8 \\ 0 & 5 & 6 & 0 & 0 \\ 0 & 4 & 0 & 3 & 0 \\ 6 & 0 & 2 & -1 & 0 \\ 9 & 0 & 0 & 0 & 6 \end{pmatrix}$$

Toto se opakuje dokud se nezpracuje poslední výřez matice velikosti 2×2.

3.2.1.2 Pseudokód algoritmu

Algorithm 4 Markowitzova strategie

Vstup: Matice $\mathbb{A}(dim, dim)$

```
1: function MARKOWITZ
2:     NNZ (nnz_row, nnz_col)
3:     for i ← 1 to A.size − 2 do
4:         min ← ∞
5:         min_row ← min_col ← i
6:         for j ← i to A.size do
7:             for k ← i to A.size do
8:                 if A (j, k) ≠ 0 then
9:                     cost ← (nnz_row (j) − 1) ∗ (nnz_col (k) − 1)
10:                end if
11:                if min > cost then
12:                    min ← cost
13:                    min_row ← j
14:                    min_col ← k
15:                end if
16:            end for
17:        end for
18:        if i ≠ min_row then
19:            swap_row (i, min_row)
20:        end if
21:        if i ≠ min_col then
22:            swap_col (i, min_col)
23:        end if
24:        update_NNZ (nnz_row, nnz_col, i)
25:    end for
26: end function
```

Funkce NNZ spočítá počty nenulových prvků na daném řádku/sloupci, funkce *update_NNZ* tyto hodnoty aktualizuje pro aktuální výřez matice využitý v další iteraci algoritmu.

3.2.1.3 Možnosti paralelizace

Možnosti paralelizace Markowitzovy strategie jsou znatelně lepší než předchozí heuristiky. Jednotlivé iterace cyklu i na sobě jsou přímo závislé, tento cyklus tedy paralelizovat nelze. Co všal lze pararelizovat jsou funkce NNZ a *update_NNZ*. Hlavně jde ale pararelizovat cyklus k. Ten je sice částečně závislý na svojí předchozí iteraci, toto však lze obejít přiřazením lokální ko-

pie každému vláknu a po skončení cyklu vybrat instanci vlákna, které našlo opravdové minimum.

Kapitola 4

Existující řešení

Tato kapitola obsahuje různé diplomové práce zabývající se podobnými tématy a již existující aplikace buďto určené přímo k řešení této problematiky, nebo obsahující algoritmy týkající se LU faktorizace.

4.1 Diplomové práce

Nejpodobnější je práce Gabriely Turcajové[9], která implementovala Doolittlův algoritmus a heuristiky Multiple Minimum Degree Ordering, Reverse Cuthill-McKee a Markowitzovu strategii. Rozdíl mezi touto prací a prací Gabriely Turcajové je inkluze paralelního zpracování.

Práce Lukáše Turčana[10] měla za úkol vytvořit řešičku řídkých soustav lineárních rovnic porovnávající různé metody řešení. Konkrétně LU rozklad, Gaussovu eliminační metodu, metodu konjugovaných gradientů a metodu bikonjugovaných gradientů.

Práce Milada Jiráska[11] měla za úkol také vytvořit řešičku řídkých soustav lineárních rovnic, avšak za pomoci Doolittlova algoritmu, Choleskyho dekompozice a Gaussovy eliminační metody. Obsaženy také byly heuristiky AMD, COLAMND a RCM.

Práce Stanislava Kusého[12] se zabývala stejným tématem jako tato práce, ale algoritmy pro LU faktorizaci byly Croutova dekompozice, Choleskyho dekompozice a QR dekompozice.

4.2 Software

R [13] R je volně dostupný software určený pro statistické výpočty. LU faktorizace je využita ve funkcích „det“ a „solve“. Dostupná je také funkce „sparseLU“, které přímo vypočítá LU faktorizaci zadané matice.

MATLAB [14] MATLAB je software určený pro práci s maticemi. LU faktorizace je využita ve funkcích „det“ která vypočte determinant matice, „inv“ která vypočte inverzi matice a „lu“ která vypočte LU faktorizaci matice.

SuperLU [15] SuperLU je software určený pro řešení soustav lineárních rovnic. Při výpočtech využívá LU faktorizaci i heuristiky pro snížení „fill-in“.

SciPy [16] SciPy je matematická knihovna pro programovací jazyk Python. Obsahuje několik algoritmů pro faktorizaci matic.

Kapitola 5

Implementace

Tato kapitola popisuje nástroje použité pro implementaci, přesnou implementaci Doolittlova algoritmu a již zmíněných heuristik.

5.1 Použité nástroje

Pro implementaci je použit jazyk *C++*. Z *C++* jsou použity kontejnery z knihovny STL, jmenovitě kontejner *vector*, *queue*, set a *multimap*, knihovna *algorithm* a knihovna *limits*, která je použita pro získání hodnoty *Epsilon*. Jako *Epsilon* se v programovacích jazycích označuje povolená odchylka při porovnávání čísel s pohyblivou desetinnou čárkou.

Z kontejneru *vector* jsou použity funkce *insert*, *size*, *reserve*, *resize*, *push_back* a *clear*. Funkce *push_back, size* a *reserve* mají složitost $\mathcal{O}(1)$. Funkce *clear* má obecně složitost $\mathcal{O}(n)$, ale pro primitivní datové typy je funkce optimalizována na $\mathcal{O}(1)$. Funkce *insert* má složitost $\mathcal{O}(n)$, kde n je počet prvků následujících po nově vloženém a funkce *resize* má složitost $\mathcal{O}(n)$, kde n je počet nově vzniklých prvků.

Z kontejneru *queue* jsou využity funkce *push, pop, empty* a *front*. Všechny mají složitost $\mathcal{O}(1)$.

Z kontejneru *set* jsou využity funkce *insert, erase* a *size*. Funkce *insert* a *erase* má složitost $\mathcal{O}(\log n)$, funkce *empty* má složitost $\mathcal{O}(1)$.

Z kontejneru *multimap* jsou využity funkce *insert* a *erase*. Implementace umožňuje využití verzí těchto funkcí s amortizovanou složitostí $\mathcal{O}(1)$.

Z knihovny *algorithm* jsou využity funkce *lower_bound, upper_bound* a *sort*. Funkce *lower_bound* a *upper_bound* mají složitost $\mathcal{O}(\log n)$, *sort* má složitost $\mathcal{O}(n \log n)$.

Pro realizaci paralelizace je využita knihovna *OpenMP*. Je také využívána k přesnému měření času.

5.2 Organizace implementace

Kompletní implementace Doolittlova algoritmu a heuristik pro snížení „fill-in“ je obsažena ve třídě *Matrix*. Tato třída reprezentuje jak vstupní tak výstupní matice a obsahuje podpůrné funkce pro operace s maticí. Mezi tyto funkce patří *get_elem* která vrátí hodnotu prvku na zadané pozici. Složitost této funkce je $\mathcal{O}(\log n)$, kde n je počet nenulových prvků v daném řádku při použití CSR a sloupci při použití CSC. Funkce *insert_elem* přidá do matice nový nenulový prvek na určenou pozici. Složitost této funkce je $\mathcal{O}(\log n + m + k)$, kde n je počet prvků na daném řádku/sloupci, m je počet prvků k aktualizaci v poli *Pointer* a k je počet prvků určených k posunu v polích *Index* a *Value*.

Třída *Matrix* obsahuje vnitřní třídu *Graph*, která reprezentuje graf využívaný v některých heuristikách. Tato třída také obsahuje podpůrné funkce, jako například hledání uzlu s nejnižším stupněm.

Podle výsledků práce Stanislava Kusého[12] jsou nejvýhodnější formáty uložení CSC a CSR, jsou proto zvoleny tyto 2 formáty. Jelikož je v Doolittlově algoritmu vstupní matice procházena jak po řádcích tak po sloupcích, tak lze zvolit jak formát CSC, tak formát CSR. V této práci byl zvolen formát CSC. Pro heuristiky je formát uložení irelevantní.

U výstupních matic však již na zvoleném formátu záleží. Matice $\mathbb{L}$ je uložena pomocí formátu CSC, jelikož je stavěna po sloupcích. Matice $\mathbb{U}$ je uložena pomocí formátu CRS, jelikož je stavěna po řádcích. Tyto volby mají sice za efekt menší efektivnost procházení matic v Doolittlově algoritmu, zabraňují ale posouvání značného množství prvků při vkládání nových prvků do matic.

5.3 Implementace algoritmů

5.3.1 Doolittlův algoritmus

Doolittlův algoritmus je implementován podle pseudokódu 2.2 s modifikacemi popsanými v 2.4.

5.3.2 Multiple Minimum Degree Ordering

Multiple Minimum Degree Ordering je implementován podle pseudokódu 3.1.1.2 s modifikacemi na 3.1.2 a 3.1.2.1.

Ovšem při implementaci se s teorií popsanou v 3.1.2.1 vyskytl problém, konkrétně se snahou zpracovávat několik sousedů smazaného uzlu najednou. Buďto se pro reprezentaci sousedů použije binární strom (tedy kontejner *set*) s kterým se ovšem stane z operace + pro přiřazování práce vláknům operace se složitostí $\mathcal{O}(n)$. Tímto se zastíní jakákoli snaha pararelizace. Pokud se pro reprezentaci použije pole (tedy kontejner *vector*), operace + bude sice $\mathcal{O}(1)$, ale z $\mathcal{O}(\log n)$ operace přidání a smazání prvku se stane operace se složitostí $\mathcal{O}(n)$, tedy znovu zastiňující jakoukoli snahu o pararelizaci. Zvolen byl tedy

způsob implementace v teoretické části popsán jako horší, avšak v praxi reálnější.

5.3.3 Reverse Cuthill-McKee

Reverse Cuthill-McKee je implementován podle pseudokódu 3.1.3.2 s modifikacemi na 3.1.4 a 3.1.4.1.

5.3.4 Markowitzova strategie

Markowitzova strategie je implementována podle pseudokódu 3.2.1.2 s modifikacemi na 3.2.1.3.

Kapitola 6

Testování

6.1 Hardware

Výsledná implementace je testována na serveru `star.fit.cvut.cz`. Na výpočetních uzlech je zde zajištěn běh pouze jednoho programu v jeden čas, běh tedy není ovlivněn vnějšími vlivy. Výpočetní uzly na serveru `star.fit.cvut.cz` obsahují následující hardware:

- 2× Intel Xeon 2620 v2 @ 2.1Ghz, Turbo 2.6Ghz

  6 jader, 12 vláken

  15MB cache

  AVX

- 32GB RAM

Na výpočetních uzlech jsou také dostupné GPU, pro účely této práce však nejsou využity.

6.2 Kompilátor

Pro kompilaci je použit kompilátor *g++*. Kompilátor je spouštěn s přepínači „*-std=c++11 -Ofast -mavx -fopenmp*“.

Přepínač „*-std=c++11*“ aktivuje standart *C++11*, který přidává různé syntaktické zkratky a nové funkce do různých knihoven. Některé již existující funkce také modifikuje či rozšiřuje. Bez tohoto přepínače je implicitní nastavení kompilátoru standardně *C++98*.

Přepínač „*-Ofast*“ je kombinací přepínačů

- *-O3*
- *-ffast-math*
- *-fno-protect-parens*

- *-fstack-arrays*

Přepínač „*-mavx*“ zaručí využívání AVX vektorových instrukcí.

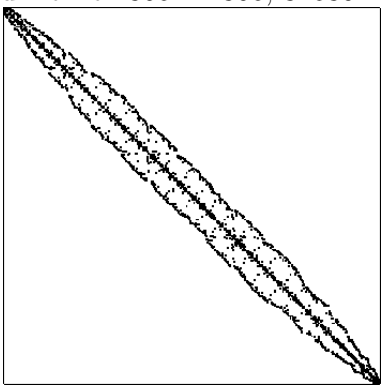
Přepínač „*-fopenmp*“ zapne v programu podporu knihovny *OpenMP*. Tato knihovna je využita pro realizaci pararelizace. Je z ní také použita funkce *omp_get_wtime()* pro přesné měření času.

6.3 Testovací data

Testovací data jsou převzata z portálu „Matrix Market“, což bezplatný portál poskytující množství různých matic. Pro testovací účely jsou vybrány řídké symetrické čtvercové matice o rozměrech zhruba 600×600 až 2000×2000.

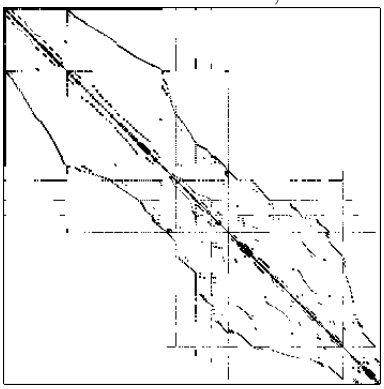
6.3.1 Testované matice

BCSSTK14 dimenze 1806×1806, 32630 nenulových prvků



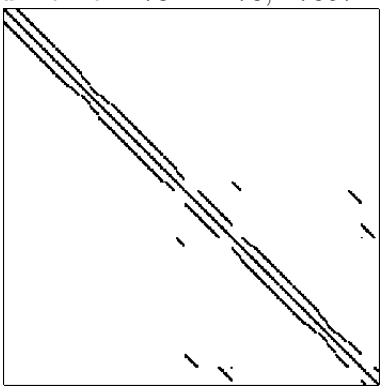
Obrázek 6.1: Matice BCSSTK14

BCSSTK08 dimenze 1074×1074, 7017 nenulových prvků



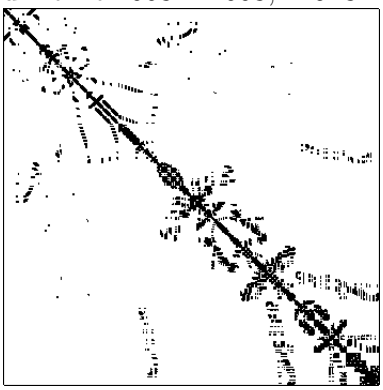
Obrázek 6.2: Matice BCSSTK08

BCSSTK11 dimenze 1473 × 1473, 17857 nenulových prvků



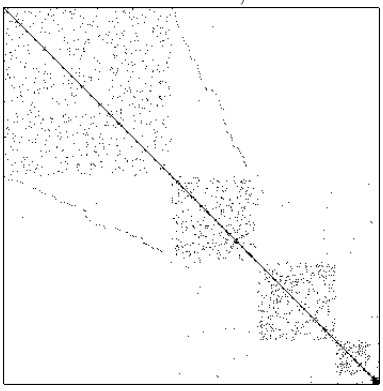
Obrázek 6.3: Matice BCSSTK11

BCSSTM13 dimenze 2003 × 2003, 42943 nenulových prvků



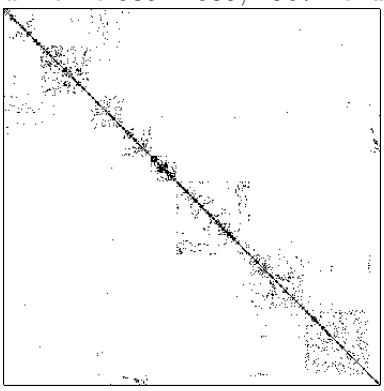
Obrázek 6.4: Matice BCSSTM13

662 BUS dimenze 662 × 662, 1568 nenulových prvků



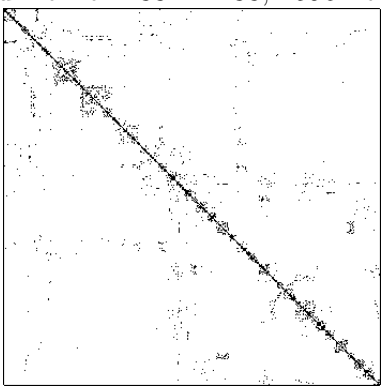
Obrázek 6.5: Matice 662 BUS

685 BUS dimenze 685 × 685, 1967 nenulových prvků



Obrázek 6.6: Matice 685 BUS

1138 BUS dimenze 1138 × 1138, 2596 nenulových prvků

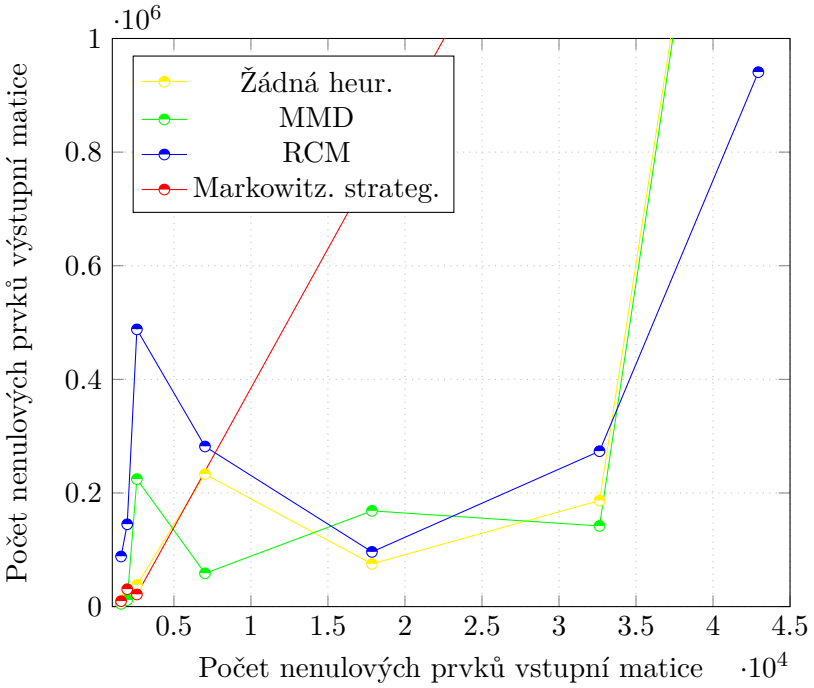


Obrázek 6.7: Matice 1138 BUS

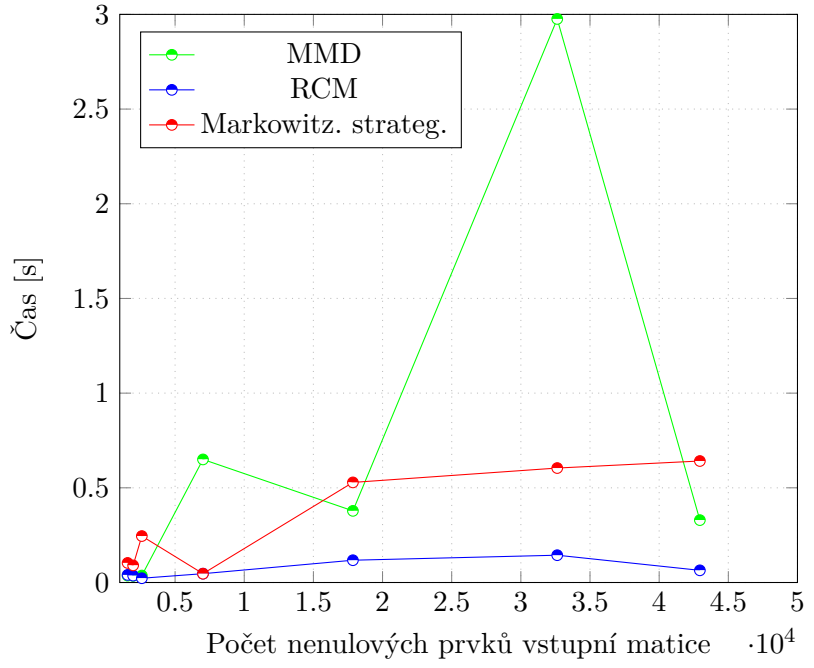
6.4 Výsledky

Z těchto grafů lze vyčíst 2 hlavní závěry. Jeden závěr je potvrzení odvozené složitosti $\mathcal{O}(n^3)$ Doolittlova algoritmu. Druhý závěr nám říká, že úspěch heuristik jak co se týče časové složitosti tak úspěchu v redukci „fill-in“ závisí silně na organizaci nenulových prvků ve vstupní matici. Je složité tedy určit nejlepší heuristiku, jejich úspěšnost se totiž liší případ od případu. V některých případech je dokonce lepší heuristiku nepoužít vůbec, jelikož počet nenulových prvků ve výsledných maticí může zásadně vzrůst.

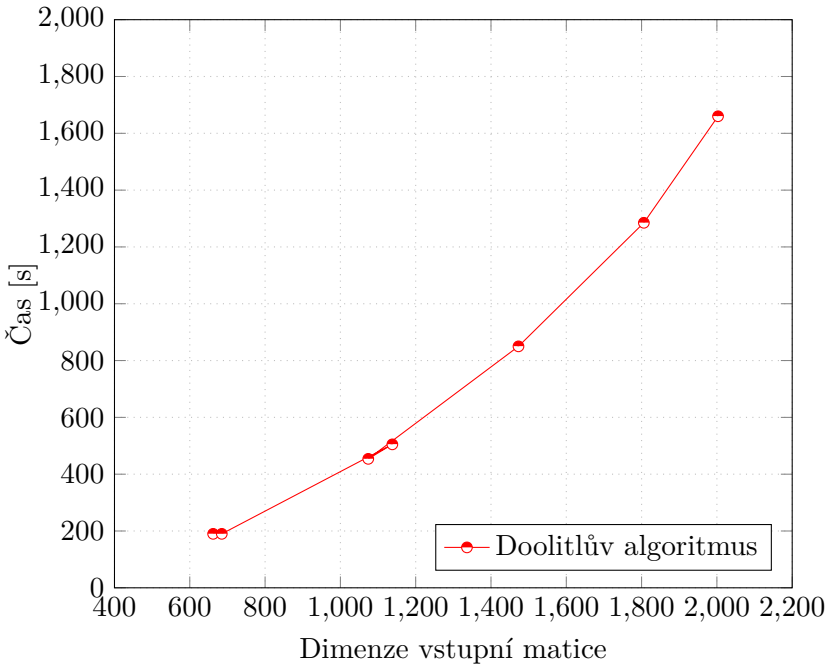
Při volbě vhodné heuristiky je tedy třeba jako první prozkoumat, která z heuristik dává průměrně nejlepší výsledky pro danou aplikace.



Obrázek 6.8: Počet nenulových prvků po LU faktorizaci



Obrázek 6.9: Časová náročnost heuristik



Obrázek 6.10: Časová náročnost Doolittlova algoritmu

Závěr

Tato práce měla za cíl seznámení se s Doolittlovým algoritmem pro LU faktorizaci matic, konkrétně řídkých matic, a heuristikami snažící se minimalizovat tak zvaný „fill-in“. V této práci byly obsaženy heuristiky Multiple Minimum Degree Ordering, Reverse Cuthill-McKee a Markowitzova strategie. Tyto algoritmy byly implementovány pararelně pomocí knihovny *OpenMP*. Zároveň byl cíl také tyto algoritmy implementovat a analyzovat. Výsledná práce tyto cíle splňuje.

Práce se příliš nezabývá faktory úspěchu heuristik v redukci „fill-inu“, jako rozšíření by tedy bylo možné algoritmy detailně prozkoumat a pokusit se vyvodit vlastnosti matic, které dané heuristice prospívají, a naopak které škodí.

Literatura

[1] Dombek, D.; Klouda, K.: Lineární algebra. *EDUX [online]*, [cit. 2017-5-15]. Dostupné z: `https://edux.fit.cvut.cz/courses/BI-LIN/_media/buildbot/lin-text.pdf`

[2] Šimeček, I.: Efektivní implementace algoritmøu. *EDUX [online]*, [cit. 2017-5-15]. Dostupné z: `https://edux.fit.cvut.cz/courses/BI-EIA/_media/lectures/volba.pdf`

[3] Matrix Market: *Matrix Market: Text File Formats*.

[4] Malík, J.; Scholtzová, J.; Suchý, O.; aj.: Algoritmy a grafy 1. *EDUX [online]*, [cit. 2017-5-15]. Dostupné z: `https://edux.fit.cvut.cz/courses/BI-AG1/lectures/start`

[5] Doolittle Decomposition of a Matrix. *College of engineering - Colorado [online]*, [cit. 2017-5-15]. Dostupné z: `http://www.engr.colostate.edu/~thompson/hPage/CourseMat/Tutorials/CompMethods/doolittle.pdf`

[6] SCIENCE, I. F. C. A. I.; VA, E. H.: The Computational Complexity of the Minimum Degree Algorithm. *Ciprian Zavoianu - weblog [online]*, prosinec 2001, [cit. 2017-5-15]. Dostupné z: `http://oai.dtic.mil/oai/oai?verb=getRecord&metadataPrefix=html&identifier=ADA398632`

[7] Tutorial: Bandwidth reduction - The CutHill-McKee Algorithm. *Ciprian Zavoianu - weblog [online]*, leden 2009, [cit. 2017-5-15]. Dostupné z: `http://ciprian-zavoianu.blogspot.cz/2009/01/project-bandwidth-reduction.html`

[8] Litovski, V.; Zwolinski, M.: *VLSI Circuit Simulation and Optimization*. Kluwer Academic Publishers, 1997, ISBN 0-412-63860-6, [cit. 2017-5-15]. Dostupné z: `http://users.ecs.soton.ac.uk/mz/CctSim/chap1_6.htm`

[9] Turcajová, G.: Efektivní LU rozklad pro řídké matice. *Bakalářská práce*, 2015.

[10] Turčan, L.: Řešice rozsáhlých soustav lineárnách rovnic. *Bakalářská práce*, 2015.

[11] Jirásek, M.: Řešice rozsáhlých soustav lineárnách rovnic. *Bakalářská práce*, 2010.

[12] Kusý, S.: Efektivní LU rozklad pro řídké matice. *Bakalářská práce*, 2015.

[13] The R Core Team: *R: A Language and Environment for Statistical Computing*. 2017, [cit. 2017-5-15]. Dostupné z: `https://cran.r-project.org/doc/manuals/r-release/fullrefman.pdf`

[14] *LU matrix factorization*. [cit. 2017-5-15]. Dostupné z: `http://www.mathworks.com/help/matlab/ref/lu.html`

[15] *SuperLU Users Guide*. 2011, [cit. 2017-5-15]. Dostupné z: `http://crd-legacy.lbl.gov/~xiaoye/SuperLU/superlu_ug.pdf`

[16] *Linear Algebra*. 2014, [cit. 2017-5-15]. Dostupné z: `https://docs.scipy.org/doc/scipy-0.14.0/reference/tutorial/linalg.html`

Příloha **A**

Seznam použitých zkratek

COLAMD Column Approximate Minimum Degree

MMD Multiple Minimum Degree

MD Minimum Degree

COO Coordinate List

CSR Compressed Sparse Row

CSC Compressed Sparse Column

Příloha **B**

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `src` ..... zdrojové kódy
  - `impl` ..... zdrojové kódy implementace
    - `matrices` ..... testovací data
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
    - `pictures` ..... obrázky
- `text` ..... text práce
  - `jurasluk_bak_prac_2017.pdf` ..... text práce ve formátu PDF