Auf der MultiFrontal-Methode basierende ILU-Zerlegungen

Zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

am Fachbereich Mathematik und
Naturwissenschaften der
Bergischen Universität Wuppertal
eingereichte

Dissertation

von

Dipl.-Math. Elton Bojaxhiu

aus Tirana, Albanien

Inhaltsverzeichnis

Einleitung

Zur Einordnung des Themas

Fast immer führt die Modellierung von Vorgängen aus Naturwissenschaft und Technik bei Simulationen im rechnerischen Kern auf die Aufgabe, ein oder mehrere lineare Gleichungssysteme der Gestalt:

$$Ax = b, \quad A \in \mathbb{R}^{n \times n}, b \in \mathbb{R}^n \tag{1}$$

zu lösen. Die Dimensionen dieser Gleichungssystemen können dabei sehr groß werden (bis zu mehrere Millionen Unbekannte, also $n \geq 10^6$). Auch wenn die modellierten Vorgänge nichtlinearer Natur sind, landet man gewöhnlich bei einem linearen Gleichungssystem der Gestalt (1), da ein Löser für nichtlineare Systeme wie z.B. das Newton-Verfahren oder die Diskretisierung partielle Differentialgleichungen als wesentlicher Teilschritt das Lösen eines linearen Gleichungssystems beinhaltet.

Das Standardverfahren zur Lösung von (1) ist die bekannte Gauß-Elimination. Sie besitzt jedoch einen Rechenaufwand von der Größenordnung n^3 und ist deshalb bei den hier interessierenden großen Werten von n bei weitem zu rechenaufwändig. Es sind deshalb seitens der Numerik im Wesentlichen zwei Alternativen entwickelt worden:

- direkte Elimination unter Berücksichtigung der Besetztheit von A ("sparse direct elimination")
- Iterationsverfahren, insbesondere Krylov-Unterraum-Methoden

Bei der "sparse direct elimination" wird davon ausgegangen, dass die Matrix A sehr viele Null-Einträge besitzt, wie dies bei den Diskretisierungen partieller

Differentialgleichungen z.B. stets der Fall ist. Unter Verwendung einer geeignet zu bestimmenden Permutation π wird dann die Matrix $A(\pi, \pi)$ mit der Standard Gauß-Elimination behandelt, allerdings unter Berücksichtigung der durch die vorhandenen Nullen möglichen Einsparungen beim Rechenaufwand. Dieser kann damit häufig drastisch gesenkt werden; das Auffinden einer geeigneten Permutation wird so zur eigentlichen (und nicht immer zufridenstellend zu lösenden) Hauptaufgabe. Die eingesetzten mathematischen Techniken stammen hier aus der Graphentheorie, angewendet auf einen, das Besetztheitmuster von A repräsentierenden, ungerichteten Graphen $G(A)$.

Bei den Iterationsverfahren wird eine Anfangsnäherung x^0 für die Lösung schrittweise verbessert; im k-ten Schritt wird eine neue Näherung x^k mit wenig Rechenaufwand aus x^{k-1} und evtl. weiteren früher Iterierten bestimmt; typischerweise erfordert jeder Schritt nur eine oder zwei vergleichsweise "billige" Matrix-Vektor-Multiplikationen mit der Matrix A. Die Frage nach dem Gesamtaufwand wird damit zur Frage nach der Gesamtzahl an Iterationen, welche benötigt wird, um eine Approximation mit akzeptabler Genauigkeit zu erhalten. Die Konvergenzgeschwindigkeit wird bestimmt durch spektrale Eigenschaften der Matrix A, entsprechend sind hier Methoden der numerischen Analysis zur Untersuchung einzusetzen.

Iterationsverfahren sind Heutzutage ohne Präkonditionierung nicht mehr denkbar. Damit wird implizit die Iteration auf der präkonditionierten Matrix (z.B.) QA ausgeführt, wobei Q eine möglichst gute Approximation der Inversen A^{-1} darstellen soll. Explizit ist in jedem Iterationsschritt ein lineares Gleichungssystem der Form $Qy = r$ zu lösen. Dies sollte also entsprechend einfach möglich sein. Für Q ist deshalb ein günstiger Kompromiss zwischen den beiden konkurrierenden Anforderungen "gute Approximation an A^{-1}" und "leichte Lösbarkeit von $Qy = r$" zu finden.

Die historisch erste und auch heute wohl noch wichtigste Präkonditionierungstechnik verwendet unvollständige LU-Zerlegungen (ILU-Zerlegungen) der Matrix A. Dies sind Darstellungen der Form

$$A = L \cdot U + R \tag{2}$$

wobei L eine linke untere und U eine rechte obere Dreiecksmatrix ist. Der Präkonditionierer ist dann $Q = (LU)^{-1} = U^{-1}L^{-1}$, und Gleichungssysteme $LUy = r$ sind aufgrund der Dreiecksgestalt von L und U problemlos lösbar. Die Kunst besteht nun darin, in der Darstellung (2) L und U einerseits so zu bestimmen, dass R *klein* ist, in dem Sinne dass A^{-1} durch $U^{-1}L^{-1}$ gut angenähert wird, dass aber andererseits die Berechnung von L und U nicht aufwändig wird und die Matrizen L und U gegenüber A nicht übermäßig auffüllen. Vom pragmatischen Gesichtspunkt aus handelt es sich dabei um die Frage, geeignete Strategien zu entwickeln, mit welchen man während der Berechnung von L und U bestimmte Zwischenresultate vernachlässigen kann (sog. *dropping*).

ILU-Zerlegungen wurden Ende der 70er Jahre eingeführt ([50]). Mit der explosionsartigen Entwicklung im Bereich der Krylov-Unterraum-Verfahren, insbesondere für unsymmetrische Systeme (GMRES, BiCGStab, QMR) Ender der 80er und Anfang der 90er Jahre, gewannen ILU-Methoden zunehmend an Bedeutung.

Wie die Arbeit organisiert ist

Die Kapitel 1 bis 6 dienen als Grundlage, um die letzten zwei Kapitel 7 und 8 zu verstehen. Ich habe versucht nicht mehr niederzuschreiben als nötig ist, um das was ich gemacht habe zu verstehen. Die Arbeit ist wie folgt organisiert:

1. Allgemeine Begrife und Definitionen über Graphen, sparse Matrizen, sparse Formate und Permutationen
2. Die Bestimmung von Transversalen, Zusammenhangskomponenten, der beidseitig irreduziblen Normalform und gewichteten Transversalen
3. Die Bestimung von symmetrischen Permutationen um das Fill-in klein zu halten
4. Beschreibung des Eliminationsbaums und der multifrontalen Methode
5. Beschreibung der meist verwendeten Krylov-Unterraum-Methoden und Präkonditionierungstechniken
6. Es werden einige Beziehungen zwischen den AINV- und ILU-Algorithmen vorgestellt. Anschließend wird die neuartige *inverse based dropping* Strategie, um Zwischenresultate zu vernachlässigen, vorgestellt
7. Es wird beschrieben, wie man aus der multifrontalen Methode, eine *neue* ILU mit *inverse based dropping* und Pivotisierung gewinnt
8. Die multifrontale Implementierung wird etwas genauer beschrieben. Es folgt eine (arithmetisch identische) zweite Implementierung, diesmal ohne multifrontale Matrizen, und deren Beschreibung. Beide Varianten werden untereinander und abschließend mit dem ILUPACK von Bollhöfer (und Saad) verglichen. Abschließend werden Parallelisierungsmöglichkeiten analysiert und getestet

Kapitel 1

Graphen, Matrizen und Permutationen

1.1 Allgemeine Definitionen

Ein **Digraph** ist ein Paar $G = (V, E)$ wobei die Knotenmenge V eine beliebige Menge und $E \subset (V \times V) \backslash \{(v, v) : v \in V\}$ die Kantenmenge ist. Typischerweise ist $V = \{1, 2, \ldots, n\}$, mit $n \in \mathbb{N}$.

Beachte, dass Loops nicht als Kanten gelten.

Ein **gewichteter Digraph** ist ein Tripel $G_g = (V, E, g)$, sodass $G = (V, E)$ ein Digraph ist und $g : E \to \mathbb{R}$ eine Gewichtfsunktion für die Kanten.

Ein **Graph** ist ein symmetrischer Digraph, d.h.

$$(\forall u, v \in V)((u, v) \in E \Rightarrow (v, u) \in E).$$

Daraus folgt, dass die Kantenrichtungen eines Graphen irrelevant sind. Beide Kanten (u, v) und (v, u) werden daher zu einer einzigen Kante $\{u, v\}$, bei der die Richtung unwichtig ist.

Ein **gewichteter Graph** ist ein Tripel $G_g = (V, E, g)$, sodass $G = (V, E)$ ein Graph ist und $g : E \to \mathbb{R}$ eine Gewichtsfunktion für die Kanten.

Sei $G = (V, E)$ ein Digraph. Der **zu G gehörige Graph** $G' = (V, E')$ ist gegeben durch $E' = \{\{u, v\} : (u, v) \in E \vee (v, u) \in E\}$.

Sei $G = (V, E)$ ein Digraph und $\overline{V} \subset V$. Der auf $\overline{V}$ **induzierte Digraph** $\overline{G} = (\overline{V}, \overline{E})$ ist gegeben durch:

$$\overline{E} = \{(u, v) \in E : u, v \in \overline{V}\}$$

Sei $G = (V, E)$ ein Graph und $\overline{V} \subset V$. Der auf $\overline{V}$ **induzierte Graph** $\overline{G} = (\overline{V}, \overline{E})$

ist gegeben durch:

$$\overline{E} = \{\{u, v\} \in E : u, v \in \overline{V}\}.$$

Seien $G = (V, E)$ und $G' = (V', E')$ zwei Digraphen. Wir sagen, dass G und G' **isomorph** sind, falls es eine bijektive Funktion $\pi : V \mapsto V'$ gibt, sodass

$$(\forall i, j \in V)((i, j) \in E \iff (\pi(i), \pi(j)) \in E')$$

und verwendet dafür die Notation $G \overset{\pi}{\cong} G'$. Analoges definiert man auch für Graphen.

Ein **bipartiter Graph** ist ein Tripel $BG = (R, C, E)$ mit Knotenmenge $R \cup C$ und Kantenmenge $E \subseteq R \times C$. Ein **gewichteter bipartiter Graph** ist ein Quadrupel $BG_g = (R, C, E, g)$, sodass $BG = (R, C, E)$ ein bipartiter Graph ist und $g : E \to \mathbb{R}$ eine Gewichtsfunktion für die Kanten.

Sei $A \in \mathbb{R}^{n \times n}$ eine Matrix. Der **zu** A **gehörige Digraph** ist $G(A) = (V, E)$ mit

$$V = \{1, 2, \ldots, n\} \text{ und}$$

$$E = \{(i, j) \in V \times V \ : \ (i \neq j) \wedge (A_{i,j} \neq 0)\}.$$

Man kann die Größen $A_{i,j}$ auch als Gewichte benutzen und den Digraphen gewichtet machen.
Ist A symmetrisch, so ist $G(A)$ ein Graph.
Der **zu** A **gehörige bipartite Graph** ist $BG(A) = (R, C, E)$ mit

$$R = \{r_1, r_2, \ldots, r_n\},\ C = \{c_1, c_2, \ldots, c_n\} \text{ und}$$
$$E = \{(r_i, c_j) \in R \times C \ : \ A_{i,j} \neq 0\}$$

Auch dieser kann durch die Größen $A_{i,j}$ gewichtet werden.

Eine Knotenteilmenge $C \subseteq V$ eines Graphen $G = (V, E)$ bildet eine **Clique**, falls $(\forall u, v \in C)(u \neq v \Rightarrow \{u, v\} \in E)$.

Sei $G = (V, E)$ ein Digraph und $u \neq w$ zwei Knoten. Ein (gerichteter) **Pfad** von u nach w ist eine Knotenfolge $u = v_0, v_1, \ldots, v_m = w$, sodass $(\forall i = 0, \ldots, m - 1)$ $(v_i, v_{i+1}) \in E$. Dabei ist m die Länge des Pfades. Falls alle Knoten dieses Pfades paarweise verschieden sind, sagt man, dass der Pfad **einfach** ist. Ähnliche Definitionen gelten auch für Graphen.

Gibt es einen Pfad von u nach w, so sagt man, dass w von u aus **erreichbar** ist.

In einem Digraph, bezeichnet ein **Zyklus** jede Knotenfolge $v_0, v_1, \ldots, v_m$, mit $m \geq 1$, sodass $(\forall i = 0, \ldots, m - 1)$ $(v_i, v_{i+1}) \in E$ und außerdem $(v_m, v_0) \in E$. Die Länge des Zyklus ist $m + 1$. Anders gesagt ist ein Zyklus ein abgeschlossener Pfad ($u = w = v_0$) der Länge $m + 1$ (≥ 2). Falls alle Knoten des Zyklus paarweise

verschieden sind, dann sagt man, dass der Zyklus **einfach** ist. Ein zyklusfreier Digraph heißt **DAG** (directed acyclic graph).

In einem Graph, bezeichnet ein **Zyklus** jede Knotenfolge $v_0, v_1, \ldots, v_m$, mit $m \geq 2$, sodass $(\forall i = 0, \ldots, m-1)$ $\{v_i, v_{i+1}\} \in E$ und noch $\{v_m, v_0\} \in E$. Die Länge des Zyklus ist $m+1$. Anders gesagt ist ein Zyklus ein abgeschlossener Pfad $(u = w = v_0)$ der Länge $m+1$ (≥ 3, im Unterschied zu Digraphen). Falls alle Knoten des Zyklus paarweise verschieden sind, dann sagt man, dass der Zyklus **einfach** ist.

Ein Graph $G = (V, E)$ heißt **zusammenhängend**, falls $\forall u \neq w$ aus V ein Pfad von u nach w existiert. Ein Digraph $G = (V, E)$ heißt zusammenhängend, falls der zugehörige Graph zusammenhängend ist.

Ein Digraph $G = (V, E)$ heißt **stark zusammenhängend**, falls $\forall u \neq w$ aus V ein (gerichtete)Pfad von u nach w existiert.

Ein **Baum** ist ein zyklusfreier zusammenhängender Graph.

Sei $G = (V, E)$ ein Graph oder Digraph, $u \in V$ und $S \subset V$ mit $u \notin S$. Wir definieren den **Reach Operator** als

$$\text{Reach}_G(u, S) = \{w \in V \backslash (\{u\} \cup S) : \text{ es gibt einen Pfad } u, \underbrace{v_1, \ldots, v_m}_{\text{aus } S \text{ mit } m \geq 0}, w \text{ von } u \text{ nach } w\}.$$

Sei $G_g = (V, E, g)$ ein gewichteter Graph oder Digraph, wobei $g \geq 0$, d.h. $g : E \to \mathbb{R}^+$. Seien u und w zwei verschiedene Knoten. Falls w von u aus nicht erreichbar ist, dann definiere den **Abstand** von u nach w als $\delta_g(u, w) = +\infty$. Sonst definiere:

$$\delta_g(u, w) = \min \left\{ \sum_{i=0}^{m-1} g(v_i, v_{i+1}) : u = v_0, v_1, \ldots, v_m = w \text{ Pfad von } u \text{ nach } w \right\}.$$

Der Abstand kann auch für ungewichtete Graphen oder Digraphen erweitert werden. Dafür definiere $g : E \to \{1\}$ d.h. gewichte jeder Kante mit 1. Statt δ_g wird hier δ geschrieben.

Sei $G = (V, E)$ ein Graph. Die **Exzentrizität** eines Knotens $u \in V$ ist gegeben durch

$$\epsilon(u) = \max_{w \in V} \delta(u, w).$$

Der **Durchmesser** eines Graphen $G = (V, E)$ ist gegeben durch

$$d(G) = \max_{v \in V} \epsilon(v).$$

Sei $G = (V, E)$ ein zusammenhängender Graph. Ein Knoten $v \in V$ heißt **peripher**, falls

$$\epsilon(u) = d(G)$$

und er heißt **pseudoperipher**, falls

$$\forall u \in V,\ \delta(u,v) = \epsilon(v) \Longrightarrow \epsilon(u) = \epsilon(v).$$

Sei $G = (V, E)$ ein Digraph und $(V_1, V_2, \ldots, V_k)$ eine Partition der Menge V. Der **Quotientendigraph** $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ ist definiert als

$$\mathcal{V} = \{V_1, V_2, \ldots, V_k\}$$

$$\mathcal{E} = \{(V_i, V_j) : (i \neq j) \wedge (\exists v_i \in V_i)(\exists v_j \in V_j)((v_i, v_j) \in E)\}.$$

Ähnlich definiert man den Quotientengraphen wenn $G = (V, E)$ ein Graph ist: $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ mit

$$\mathcal{V} = \{V_1, V_2, \ldots, V_k\}$$

$$\mathcal{E} = \{\{V_i, V_j\} : (i \neq j) \wedge (\exists v_i \in V_i)(\exists v_j \in V_j)(\{v_i, v_j\} \in E)\}.$$

Sei $G = (V, E)$ ein Graph. Eine Knotenmenge $W \subset V$ heißt **unabhängig**, falls der auf W induzierter Graph keinerlei Kanten besitzt.

Eine Matrix $A \in \mathbb{R}^{m \times n}$ heißt **nichtnegativ** (bzw. **positiv**), falls $\forall i, j \in \{1, \ldots, n\}$, $A_{i,j} \geq 0$ (bzw. $A_{i,j} > 0$). In Kurzform schreibt man $A \geq 0$ (bzw. $A > 0$).

Eine Matrix $A \in \mathbb{R}^{n \times n}$ heißt **M-Matrix**, falls folgende Bedingungen erfüllt sind

1. $\forall i = 1, \ldots, n$, $A_{i,i} > 0$
2. $\forall i \neq j$ aus $\{1, \ldots, n\}$, $A_{i,j} \leq 0$
3. A ist nichtsingulär
4. $A^{-1} \geq 0$

Die erste Eigenschaft ist eigentlich eine Folgerung der anderen 3 Eigenschaften.

1.2 Sparse Formate für Vektoren

Für einen Vektor $x \in \mathbb{R}^n$, definiert man dessen **Struktur** als:

$$\text{Struct}(x) = \{j : x_j \neq 0\}.$$

Besitzt x "zu viele" Nulleinträge, d.h. x ist "sparse", so kann das ausgenutzt werden, um den Speicherverbrauch von x zu reduzieren: Speichere x in komprimierter Form als ein Paar (ind_x, val_x). Beide Vektoren, ind_x und val_x, besitzen

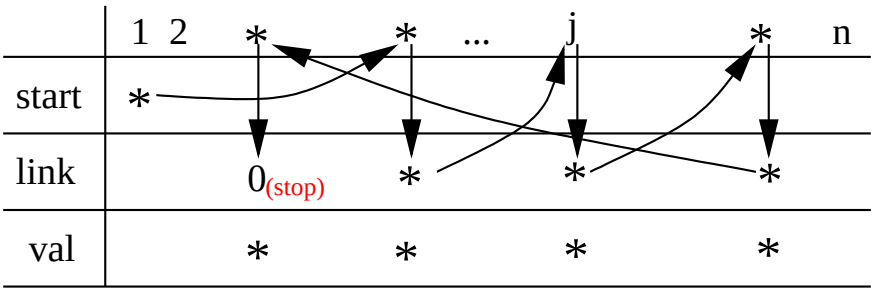
die Länge $\text{nnz}(x) = |\text{Struct}(x)|$. Als Menge betrachtet, ist ind_x identisch mit $\text{Struct}(x)$ und $x(ind_x) = val_x$. Die Indizes in ind_x müssen nicht geordnet sein. Sollten sie es aber sein, so heißt das Format *geordnet*. Die Dimension n gehört auch zum Format, aber implizit. Das beschriebene Format bezeichnet man als **gepacktes** Format.

Ein anderes Format ist das **Listenformat** $(start, link, val)$. $start$ ist eine Startindex, $link$ und val sind Vektoren der Länge n. Wie stellen $(start, link, val)$ den Vektor x dar? Definiere den Vektor ind nach dem folgenden Algorithmus:

Algorithmus 1.1 generiere ind

Beschreibung: Input: $start$, $link$; Output: ind

1: $j = start$, $i = 0$
2: **while** $j \neq 0$ **do**
3: $\quad i = i + 1$
4: $\quad ind(i) = j$
5: $\quad j = link(j)$
6: **end while**



Nur wenn das Paar $(ind, val(ind))$ ein gepacktes Format für x ist, ist $(start, link, val)$ ein Listenformat.

Z.B. sei $x = (2.1, 2.2, 0, 2.4, 0)$.

- (ind_x, val_x) mit $ind_x = (4, 1, 2)$ und $val_x = (2.4, 2.1, 2.2)$ ist eine gepackte Format
- $(start, link, val)$ mit $start = 4$, $link = (0, 1, *, 2, *)$ und $val = (2.4, 2.2, *, 2.1, *)$ ist eine Listenformat ($*$ steht für beliebiege Werte)

Listenformate sind **nicht** attraktiv für einen einzelnen sparsen Vektor, sondern für mehrere, sagen wir m, sparse Vektoren mit paarweise disjunkten Strukturen. Entsprechend muss $start$ zu eine Vektor der Länge m angepasst werden (siehe Listenformate für Matrizen).

1.3 Sparse Matrizen

Definiere die **Struktur** und **nnz** von A als:

$$\text{Struct}(A) = \{(i,j) : A_{i,j} \neq 0\},\ \text{nnz}(A) = |\,\text{Struct}(A)|.$$

Eine klare Definition für sparse Matrizen gibt es nicht. Zwei Beschreibungsmöglichteiten wären:

- Die $n \times n$ Matrix A ist sparse, falls $\text{nnz}(A) = \mathcal{O}(n)$
- (nach [30])Die $n \times n$ Matrix A ist sparse, falls es sich lohnt, die "vielen" Nullen auszunutzen, um den Aufwand von bestimmten Algorithmen zu reduzieren.

Als Beispielmatrix wird hier die Matrix

$$A = \begin{pmatrix} 1.1 & 0 & 0 & 1.4 & 0 \\ 2.1 & 2.2 & 0 & 2.4 & 0 \\ 0 & 0 & 3.3 & 0 & 3.5 \\ 0 & 4.2 & 0 & 4.4 & 0 \\ 5.1 & 0 & 5.3 & 5.4 & 5.5 \end{pmatrix}$$

dienen.

Das naheliegendste Format ist das **Koordinatenformat** (rn, cn, val). Alle drei Vektoren besitzen die Länge $\text{nnz}(A)$; $(rn(k), cn(k))_{k=1:\text{nnz}(A)}$ ist als Menge betrachtet identisch mit $\text{Struct}(A)$ und $(\forall k = 1, \ldots, \text{nnz}(A))\ (A_{rn(k),cn(k)} = val(k))$.

Für unsere Beispielmatrix ist das Tripel (rn, cn, val):

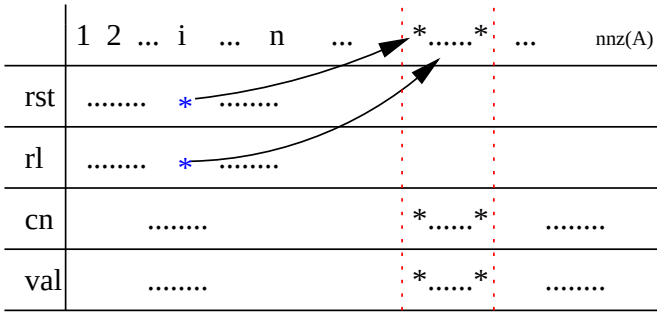
	1	2	3	4	5	6	7	8	9	10	11	12	13
rn	3	3	2	1	2	2	5	4	5	4	1	5	5
cn	3	5	4	4	2	1	1	4	4	2	1	5	3
val	3.3	3.5	2.4	1.4	2.2	2.1	5.1	4.4	5.4	4.2	1.1	5.5	5.3

ein Koordinatenformat.

Ein anderes Format ist das zeilenweise[1] **gepackte** Format (rst, rl, cn, val). rst und rl sind Vektoren der Länge n, während cn und val Vektoren der Länge $\text{nnz}(A)$ sind. Für alle $i = 1, \ldots, n$, ist:

$$(cn(rst(i) : rst(i) + rl(i) - 1), val(rst(i) : rst(i) + rl(i) - 1))$$

[1] spaltenweise definiert man ähnlich



ein gepacktes Format für die i-te Zeile.

Ein gepacktes Format heißt *halbgeordnet*, falls rst, als Folge betrachtet, aufsteigend ist. Ein gepacktes Format heißt *geordnet*, falls für alle $i = 1, \ldots, n$, $cn(rst(i) : rst(i) + rl(i) - 1)$ aufsteigend ist. Eine gepacktes Format heißt *voll geordnet*, falls es halbgeordnet und geordnet ist. Für den letzten Fall sollte erwähnt werden, dass es dann möglich ist, den Speicherverbrauch noch mehr zu reduzieren: rst sollte Länge $n+1$ besitzen, mit $rst(n+1) = \mathrm{nnz}(A)$, rl wird überflüssig (weil $rl(i) = rst(i+1) - rst(i)$), und die Zeile i gegeben ist durch

$$(cn(rst(i) : rst(i+1) - 1), val(rst(i) : rst(i+1) - 1)) .$$

Viele Softwarepakete verlangen als Input eine Matrix im voll geordneten gepackten Format. Während dieselbe Matrix auf viele Weisen im halbgeordneten oder geordneten gepackte Formate dargestellt werden kann, ist das voll geordnete gepackte Format eindeutig.

Für das gepackte Format gibt es die interessante Operation "trp"([54]). Sie bildet mit Aufwand $\mathcal{O}(n+\mathrm{nnz}(A))$ das voll geordnete gepackte Format der transponierten Matrix A^T. Wendet man diese Operation doppelt ("doptrp") an, so erhält man das voll geordnete gepackte Format der Matrix selbst. Für symmetrische Matrizen reicht eine einzige "trp"-Operation aus.

Das Quadrupel (rst, rl, cn, val)

	1	2	3	4	5	6	7	8	9	10	11	12	13
rst	6	1	8	4	10								
rl	2	3	2	2	4								
cn	2	1	4	4	2	4	1	3	5	4	3	5	1
val	2.2	2.1	2.4	4.4	4.2	1.4	1.1	3.3	3.5	5.4	5.3	5.5	5.1

ist ein gepacktes Format für unsere Beispielmatrix.

Ein flexibles Format ist das (einfach verkettete) zeilenweise[2] **Listenformat** $(rst, link, cn, val)$. rst ist ein Vektor der Länge n, während $link$, cn und val Vektoren der Länge $\mathrm{nnz}(A)$ sind. Wie stellt dieses Quadrupel die i-te Zeile dar? Definiere die Vektoren $ind^{(i)}, val^{(i)}$ nach dem folgenden Algorithmus:

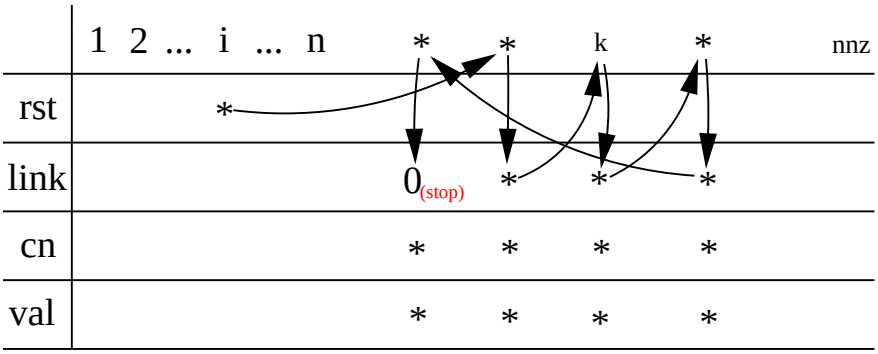
[2] spaltenweise definiert man ähnlich

Algorithmus 1.2 generiere $ind^{(i)}$ und $val^{(i)}$

Beschreibung: Input: $i, rst, link, cn, val$; Output: $ind^{(i)}, val^{(i)}$

1: $k = rst(i)$, $j = 0$
2: **while** $k \neq 0$ **do**
3: $\quad j = j + 1$
4: $\quad ind^{(i)}(j) = cn(k)$
5: $\quad val^{(i)}(j) = val(k)$
6: $\quad k = link(k)$
7: **end while**

Die Ausgabe $(ind^{(i)}, val^{(i)})$ ist ein gepacktes Format für die i-te Zeile. Folgende Abbildung veranschaulicht die Entstehung der i-ten Zeile:



Das Quadrupel $(rst, link, cn, val)$:

	1	2	3	4	5	6	7	8	9	10	11	12	13
rst	6	1	8	4	10								
$link$	2	3	0	5	0	7	0	9	0	11	12	13	0
cn	2	1	4	4	2	4	1	3	5	4	3	5	1
val	2.2	2.1	2.4	4.4	4.2	1.4	1.1	3.3	3.5	5.4	5.3	5.5	5.1

ist ein Listenformat für unsere Beispielmatrix.

Noch flexibler ist das doppelt verkettete Listenformat. Es wird normalerweise als ein "wrapper" um ein Koordinatenformat gelegt.

Das gepackte Format erlaubt am schnellsten Zugriffe auf die Einträge einer bestimmten Zeile. Beim Koordinatenformat ist dies sehr aufwändig. Beim (doppelt verketteten) Listenformaten ist es einfach möglich, Einträge hinzuzufügen oder zu entfernen.

Es ist möglich, mit Aufwand $\mathcal{O}(n+\mathrm{nnz}(A))$ zwischen verschiedenen Formaten zu konvertieren.

Zuletzt sei A eine $n \times n$ Matrix, welche eine **nullfreie** Diagonale besitzt. Sei $A = LDU$ ihre LDU-Zerlegung[3]. Sei $F = L + U$ die sog. **Fill-in Matrix**. Alle Einträge $(i, j) \in \text{Struct}(F)$, die nicht zu $\text{Struct}(A)$ gehören, heißen **Fill-in**. Die Größe

$$\text{FillIn}(A) = \frac{\text{nnz}(L) + \text{nnz}(U) - n}{\text{nnz}(A)}$$

ist der sog. **Fill-in-Faktor**, ein Maß für das Fill-in.

1.4 Permutationen auf Matrizen und Graphen

1.4.1 Permutationen auf Matrizen

Sei C eine beliebige $(m \times n)$ Matrix, π,τ Permutationen auf $\{1, \ldots, m\}$ und ψ,φ Permutationen auf $\{1, \ldots, n\}$.

Definition 1.1 *Definiere die $(m \times n)$ Matrix $D = C(\pi, \psi)$ durch*

$$\forall (i, j) \in \{1, \ldots, m\} \times \{1, \ldots, n\},\ D_{i,j} = C_{\pi(i),\psi(j)}$$

Eine noch allgemeinere Definition wäre:

Definition 1.2 *Seien I, J (erlaubte) Zeilen- bzw. Spalten-Indexfolgen, d.h.*

$$\forall i = 1, \ldots, |I|,\ I_i \in \{1, \ldots, m\} \quad \textit{und}$$

$$\forall j = 1, \ldots, |J|,\ J_j \in \{1, \ldots, n\}$$

Die $(|I| \times |J|)$ Matrix $D = C(I, J)$ ist definiert durch

$$\forall (i, j) \in \{1, \ldots, |I|\} \times \{1, \ldots, |J|\},\ D_{i,j} = C_{I_i,J_j}$$

Mit $\circ$ bezeichnen wir die gewöhnliche Komposition der Funktionen $f \circ g = f(g)$. Folgende Lemmata sind sehr einfach zu beweisen.

Lemma 1.1 *Seien τ, φ Permutationen der $1 : m$ bzw. $1 : n$. Es gilt:*

$$D(\tau, \varphi) = C(\pi \circ \tau, \psi \circ \varphi)\ \ (= C(\pi(\tau), \psi(\varphi)))\,.$$

Sei η die Identitätspermutation und I die Einheitsmatrix (implizite Ordnung). Wenn es geht, werden wir η einfach durch ":" ersetzten (wie in Matlab).

[3] existiert immer, vorausgesetzt es entstehen keine zufälligen Nullen

Lemma 1.2

$$I(\tau, \tau) = I$$

Definition 1.3 *Definiere die Permutationsmatrix P_π als:*

$$P_\pi = I(\pi, \eta) = I(\pi, :)$$

P_π gewinnt man nach der Regel "Zeile i von P_π = Zeile $\pi(i)$ von I".

Aus $P_\pi = I(\pi, \eta) = I(\pi(\eta), \pi(\pi^{-1})) = I(:, \pi^{-1})$ folgt, dass P_π auch nach der Regel "Spalte j von P_π = Spalte $\pi^{-1}(j)$ von I" gewonnen werden kann.

Lemma 1.3

$$P_\pi^{-1} = P_\pi^T = P_{\pi^{-1}}$$
$$(P_{\pi(\tau)} =)\ P_{\pi\circ\tau} = P_\tau P_\pi$$

Lemma 1.4

$$C(\pi, :) = P_\pi C$$
$$C(:, \psi) = C P_\psi^T$$
$$C(\pi, \psi) = P_\pi C P_\psi^T$$

Lemma 1.5 *Seien $\pi_1, \pi_2, \ldots, \pi_k$ Permutationen (Ordnung m). Es gilt*

$$P_{\pi_1\circ\pi_2\circ\ldots\circ\pi_{k-1}\circ\pi_k} = P_{\pi_k} P_{\pi_{k-1}} \ldots P_{\pi_2} P_{\pi_1}$$

Aus

$$P_{\pi_k} \ldots P_{\pi_1} C = P_{\pi_1\circ\ldots\circ\pi_k} C$$

folgt: wendet man die (Zeilen) Permutationen $\pi_1, \pi_2, \ldots, \pi_k$ nacheinander (in dieser Reihenfolge) auf die Matrix C an, dann ist das gleich als ob man die (Zeilen) Permutation $\pi_1 \circ \pi_2 \circ \ldots \circ \pi_k$ auf C angewandt hat.

Beispiele:
Sei $\pi = (2, 4, 5, 1, 3)$ und $\psi = (1, 4, 5, 3, 2)$. Dann ist $\psi^{-1} = (1, 5, 4, 2, 3)$ und

$$P_\pi = I(\pi, :) = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix},\ P_\psi = I(\psi, :) = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{pmatrix},\ P_\psi^T =$$
$$= P_{\psi^{-1}} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix},\ A(\pi, :) = \begin{pmatrix} 2.1 & 2.2 & 0 & 2.4 & 0 \\ 0 & 4.2 & 0 & 4.4 & 0 \\ 5.1 & 0 & 5.3 & 5.4 & 5.5 \\ 1.1 & 0 & 0 & 1.4 & 0 \\ 0 & 0 & 3.3 & 0 & 3.5 \end{pmatrix} = P_\pi A,$$

$$A(:,\psi) = \begin{pmatrix} 1.1 & 1.4 & 0 & 0 & 0 \\ 2.1 & 2.4 & 0 & 0 & 2.2 \\ 0 & 0 & 3.5 & 3.3 & 0 \\ 0 & 4.4 & 0 & 0 & 4.2 \\ 5.1 & 5.4 & 5.5 & 5.3 & 0 \end{pmatrix} = AP_\psi^T,\ A(\pi,\psi) = \begin{pmatrix} 2.1 & 2.4 & 0 & 0 & 2.2 \\ 0 & 4.4 & 0 & 0 & 4.2 \\ 5.1 & 5.4 & 5.5 & 5.3 & 0 \\ 1.1 & 1.4 & 0 & 0 & 0 \\ 0 & 0 & 3.5 & 3.3 & 0 \end{pmatrix}$$
$= P_\pi A P_\psi^T$.

1.4.2 Permutationen und Vektoren

Im Falle, dass C eine $(m \times 1)$ Matrix ist, ergibt sich:

Lemma 1.6 *Sei x ein Spaltenvektor $x \in \mathbb{R}^{m\times 1}$ (oder $\mathbb{C}^{m\times 1}$). Dann gilt $x(\pi) = P_\pi x$. Sei $y = x(\pi)$ und $z = y(\tau)$. Dann ist $z = x(\pi(\tau))$.*

1.4.3 Permutationen und sparse Formate

Sei A eine sparse Matrix $n \times n$ (hier ist $m = n$). Sei $B = A(\pi,\psi)$. Es ist möglich, aus einem sparsen Format für A, ein sparses Format für B zu bestimmen:

- Falls $(rnA, cnA, valA)$ ein Koordinatenformat für A ist, dann ist $(rnB, cnB, valB)$, mit:

  $$\pi(rnB) = rnA,\ \psi(cnB) = cnA,\ valB = valA$$

  ein Koordinatenformat für B.

- Falls $(rstA, rlA, cnA, valA)$ ein (zeilenweise) gepacktes Format für A ist, dann ist $(rstB, rlB, cnB, valB)$, mit:

  $$rstB = rstA(\pi),\ rlB = rlA(\pi),\ \psi(cnB) = cnA,\ valB = valA$$

  ein (zeilenweise) gepacktes Format für B.

- Falls $(rstA, linkA, cnA, valA)$ ein (zeilenweise) Listenformat für A ist, dann ist $(rstB, linkB, cnB, valB)$, mit:

  $$rstB = rstA(\pi),\ linkB = linkA,\ \psi(cnB) = cnA,\ valB = valA$$

  ein (zeilenweise) Listenformat für B.

1.4.4 Renumerierung der Knoten eines Graphen

Sei A eine $n \times n$ (wenigstens strukturell) symmetrische Matrix. Sei $G = (V, E)$ ihr zugehöriger (ungerichteter) Graph. Üblicherweise werden die Permutationen anhand von G bestimmt, mittels einer oder mehrerer Renumerierungen. Dadurch erhält man einen neuen Graphen $G' = (V', E')$. Wie hat sich A nach der Renumerierung geändert? D.h. finde die Matrix B, welche dem neuen Graphen $G' = (V', E')$ entspricht. Jeder, der schon mal mit Knotenrenumerierungen zu tun hatte, wird eine Zweideutigkeit bei diese Problem bemerkt haben. Eins ist klar, $V' = V = \{1, 2, \ldots, n\}$.

Sei $\sigma = (\sigma_1, \sigma_2, \ldots, \sigma_n)$ eine (Knoten) Renumerierung.

Man unterscheidet zwei Fälle des Renummerierung:

Fall 1:

$$\sigma = \begin{pmatrix} \sigma_1 & \sigma_2 & \ldots & \sigma_n \\ \downarrow & \downarrow & \cdots & \downarrow \\ 1 & 2 & \ldots & n \end{pmatrix} \quad \left(\downarrow = \begin{pmatrix} \text{kommt über} \\ \text{den Knoten } * \text{ von } G \end{pmatrix} \right)$$

über den Knoten 1 von G kommt σ_1
über den Knoten 2 von G kommt σ_2
.
.
über den Knoten n von G kommt σ_n
(**Interpretation**: der Knoten i von G wird der Knoten σ_i von G')

Fall 2: „normaler Fall“

$$\sigma = \begin{pmatrix} 1 & 2 & \ldots & n \\ \downarrow & \downarrow & \cdots & \downarrow \\ \sigma_1 & \sigma_2 & \ldots & \sigma_n \end{pmatrix} \quad \left(\downarrow = \begin{pmatrix} \text{kommt über} \\ \text{den Knoten } * \text{ von } G \end{pmatrix} \right)$$

die Zahl 1 kommt über den Knoten σ_1 von G
die Zahl 2 kommt über den Knoten σ_2 von G
.
.
die Zahl n kommt über den Knoten σ_n von G
(**Interpretation**: der Knoten σ_i von G wird der Knoten i von G')

Für den **Fall 1**:

$(\sigma_i, \sigma_j) \in E' \Longleftrightarrow (i, j) \in E$, oder, anders geschrieben
$B_{\sigma_i, \sigma_j} = A_{i,j} \Longleftrightarrow B(\sigma, \sigma) = A \Longleftrightarrow B = A(\sigma^{-1}, \sigma^{-1})$

Für den **Fall 2**:

$(i, j) \in E' \Longleftrightarrow (\sigma_i, \sigma_j) \in E$, oder, anders geschrieben
$B_{i,j} = A_{\sigma_i, \sigma_j} \Longleftrightarrow B = A(\sigma, \sigma)$

Kapitel 2

B-reduzible Normalform und gewichtete Transversalen

2.1 Transversalen

Sei A eine $n \times n$ Matrix. Führt man die Gaußelimination ohne Pivotsuche oder höchstens Pivotsuche auf der Hauptdiagonale aus, so sollte gesichert sein, dass es keine Nullen auf der Hauptdiagonale gibt[1]. D.h. es wird eine (Spalten) Permutation π gesucht, so dass $(\forall i = 1, \ldots, n)\ A_{i,\pi(i)} \neq 0$. In diesen Fall besitzt die Matrix $B = A(:, \pi)$ eine nullfreie Diagonale. Es ist egal, ob man Spalten- oder Zeilenpermutation betrachtet, denn es ist $A(:, \pi) = B$ mit nullfreier Diagonale genau dann, wenn $A(\pi^{-1}, :) = B(\pi^{-1}, \pi^{-1})$ mit nullfreier Diagonale ist.

Erinnerung: Sei π eine Permutation aus P_n (die Menge alle Permutationen aus $\{1, \ldots, n\}$). Ein Paar $i < j$ heißt **Inversion**, falls $\pi(i) > \pi(j)$. Ist deren gesamte Anzahl gerade, so heißt π *gerade*, sonst *ungerade*. So definiert man die **Parität** einer Permutation.

Satz 2.1 *Ist A nichtsingulär, so gibt es eine Spaltenpermutation π, so dass $(\forall i = 1, \ldots, n)\ A_{i,\pi(i)} \neq 0$.*

Beweis: Da $\det(A) = \sum_{\pi \in P_n} (-1)^{\text{Parität}(\pi)} A_{1,\pi(1)} \cdot \ldots \cdot A_{n,\pi(n)} \neq 0$ (LA Kurs), gibt es mindestens eine Permutation $\pi \in P_n$, so dass $A_{1,\pi(1)} \ldots A_{n,\pi(n)} \neq 0$. Das ist die gewünschte Permutation. □

Eine solche Permutation heißt **Transversale**. Eine Transversale kann alterna-

[1] Es geht um strukturelle Nullen, denn zufällige Nullen sind ausgeschlossen.

tiv, sog. **Matchingform**, als eine Menge von n Paaren $T = \{(i_1, j_1), \ldots, (i_n, j_n)\}$ dargestellt werden, wobei $\{i_1, \ldots, i_n\} = \{j_1, \ldots, j_n\} = \{1, \ldots, n\}$ und $(\forall k = 1, \ldots, n)\ A_{i_k, j_k} \neq 0$.

Eine Matrix heißt **strukturell singulär**, falls sie singulär bleibt für jede Belegung ihrer Struktur mit Werten.

Satz 2.2 *Eine Matrix besitzt keinen Transversale dann und nur dann wenn sie strukturell singulär ist. (Folgt aus* $\det(A) = \sum_{\pi \in P_n} (-1)^{Parität(\pi)} A_{1,\pi(1)} \ldots A_{n,\pi(n)}$*.)*

Um eine Transversale zu finden, gibt es zwei bekannte Möglichkeiten:

- den Algorithmus von Hall ([27],[26]) mit Aufwand $\mathcal{O}(n \cdot nnz(A))$
- den Algorithmus von Hopcroft-Karp ([42]) mit Aufwand $\mathcal{O}(\sqrt{n}(n+nnz(A)))$

Duff benutzt den ersten Algorithmus, um eine Transversale zu berechnen. Sein Code ist bekannt geworden unter den Namen MC21 und ist Teil des HSL (Harwell Subroutines Library, [25]). Da ich den Algorithmus von Hopcroft-Karp benutzt habe, werde ich diesen nun beschreiben. Für die Beweise verweise ich auf ([42]).

Definition 2.1 *Sei $G = (V, E)$ ein Graph. Eine Menge $M \subseteq E$ heißt **Matching** bezüglich G, falls je zwei verschiedene Kanten aus M keine gemeinsamen Endknoten haben.*

- Ein Matching M heißt **maximal**, falls es kein anderes Matching N gibt mit $|M| < |N|$.
- Ein Knoten v heißt **saturiert** bezüglich M, falls er als Endknote einer Kante aus M dient.
- Ein Knoten v heißt **frei** bezüglich M, falls er nicht saturiert bezüglich M ist.
- Ein Matching M heißt **perfekt**, falls es keine freien Knoten bezüglich M gibt.

Jede $(n \times n)$ Matrix A ist in natürlicher Weise ein (gewichteter) bipartite Graph $BG(A) = (BV, BE)$ mit Knoten $R_{ows} = \{r_1, \ldots, r_n\}$, $C_{ols} = \{c_1, \ldots, c_n\}$, und Kanten $BE = \{\{r_i, c_j\} | A_{i,j} \neq 0\}$. Das Transversalenproblem einer Matrix A ist äquivalent mit dem perfekten Matching-Problem ihres bipartiten Graphs $BG(A)$. Ist A strukturell nicht singulär, so sind perfekt und maximal äquivalent.

Definition 2.2 *Sei $G = (V, E)$ ein Graph und M eine Matching. Ein Pfad $p = (v_1, v_2, \ldots, v_{2k})$ heißt zunehmender Pfad bezüglich M, falls :*

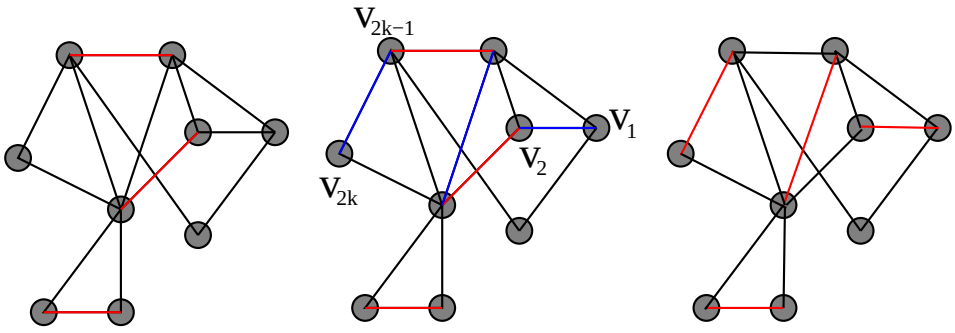
- v_1 *und* v_{2k} *freie Knoten und* $v_2, v_3, \ldots, v_{2k-1}$ *saturierte Knoten sind*
- $\{v_2, v_3\},\ \{v_4, v_5\}, \ldots, \{v_{2k-2}, v_{2k-1}\} \in M$

Manchmal wird im Folgenden der Pfad p einfach seine Kantenmenge repräsentieren, abhängig vom Kontext. Sei der Operator $\triangle$ die *symmetrische Differenz* zweier Mengen $A \triangle B = (A \backslash B) \cup (B \backslash A)$. Die folgenden Sachverhalte sind elementar zu beweisen:

Lemma 2.1 *Seien* $A, B, C \subseteq Y$*. Es gilt:*

- $A \triangle B = (A \cup B) \setminus (A \cap B) = (A \cap B^c) \cup (A^c \cap B)$
- $A \triangle B = B \triangle A$
- $(A \triangle B) \triangle C = A \triangle (B \triangle C)$
- $A \triangle (A \triangle B) = B$

Lemma 2.2 *Sei* M *ein Matching bezüglich* $G = (V, E)$ *und* p *ein zunehmender Pfad bezüglich* M*. Dann ist* $M \triangle p$ *ein Matching der Größe* $|M \triangle p| = |M| + 1$*.*



Satz 2.3 *Seien* M *und* N *zwei Matchings, so dass* $|M| < |N|$*. Dann gibt es in* $(M \triangle N)$ *mindestens* $(|N| - |M|)$ *knotendisjunkte zunehmende Pfade bezüglich* M*.*

Korollar 2.1 *Ein Matching* M *ist maximal dann und nur dann wenn es keinen zunehmenden Pfad bezüglich* M *gibt.*

Ein allgemeiner Algorithmus, um ein maximales Matching zu berechnen, sieht so aus:

Algorithmus 2.1 Finde ein maximales Matching

Beschreibung: Input: Anfangsmatching M_0 (könnte $\emptyset$ sein); Output: maximales Matching

1: $i = 0$
2: **while** M_i nicht maximal **do**
3: sei p_i ein zunehmender Pfad bezüglich M_i
4: $M_{i+1} = M_i \triangle p_i$
5: $i = i + 1$
6: **end while**

Das letzte M_i wird maximal sein.

Satz 2.4 ([42]) *Seien M und N zwei Matchings der Größe r bzw. s, wobei $s > r$. Dann gibt es einen zunehmenden Pfad bezüglich M der Länge $\leq 2\lfloor \frac{r}{s-r} \rfloor + 1$.*

Die kürzesten zunehmenden Pfade bezüglich eines Matchings sind von besonderer Wichtigkeit.

Satz 2.5 *Sei M ein Matching und p ein kürzester zunehmender Pfad bezüglich M. Sei p' ein zunehmender Pfad bezüglich $M \triangle p$. Dann gilt:*

$$|p'| \geq |p| + 2|p \cap p'|$$

Würde man statt beliebiger zunehmender Pfade kürzeste zunehmende Pfade verlangen, sähe der allgemeine Algorithmus so aus:

Algorithmus 2.2 Finde ein maximales Matching

Beschreibung: Input: Anfangsmatching M_0; Output: maximales Matching

1: $i = 0$
2: **while** M_i nicht maximal **do**
3: sei p_i ein kürzester zunehmender Pfad bezüglich M_i
4: $M_{i+1} = M_i \triangle p_i$
5: $i = i + 1$
6: **end while**

Satz 2.5 besagt, dass $|p_0| \leq |p_1| \leq |p_2| \leq \ldots$. Man kann sogar zeigen dass:

Satz 2.6 *Ist im Algorithmus (2.2) $|p_k| = |p_j|$ für $k < j$, dann sind die Pfade $p_k, p_{k+1}, \ldots, p_{j-1}, p_j$ (paarweise) knotendisjunkt und von gleicher Größe.*

Also ist die Folge $p_0, p_1, p_2, \ldots$ in Gruppen von konsekutiven Pfaden gleicher Größe partitioniert, und die Pfade einer Gruppe sind paarweise knotendisjunkt. Folgender Satz gibt eine obere Grenze für die Gesamtzahl diese Gruppen.

Satz 2.7 *Sei s die Größe eines maximalen Matchings. Die Anzahl der oben erwähnten Gruppen ist höhstens $2\lfloor\sqrt{s}\rfloor + 1$.*

Unter Verwendung der letzten zwei Sätze, kann der allgemeine Algorithmus (2.2) wie folgt modifiziert werden:

Algorithmus 2.3 Hopcroft-Karp : Finde ein maximales Matching

Beschreibung: Input: Anfangsmatching M; Output: maximale Matching

1: **while** M nicht maximal **do**
2: Finde eine maximale Menge $\{p_1, p_2, \ldots, p_t\}$ knotendisjunkter kürzester zunehmender Pfade bezüglich M {alle Pfade haben gleiche Größe}
3: $M = M \triangle p_1 \triangle p_2 \triangle \ldots \triangle p_t$
4: **end while**

Satz 2.7 zeigt, dass die *while* Schleife höchstens $\mathcal{O}(\sqrt{|V|})$ Mal durchlaufen wird, da $s \leq |V|/2$ gilt. Für *bipartite Graphen* sind die Zeilen 2 und 3 in $\mathcal{O}(|V| + |E|)$ Zeit zu schaffen[2]. In diesen Fall hat man einen Gesamtaufwand von $\mathcal{O}\left(\sqrt{|V|}\,(|V| + |E|)\right)$.

Sei nun A eine $(n \times n)$ Matrix. Der bipartite Graph von A, $BG(A)$, hat $2n$ Knoten und $nnz(A)$ Kanten. Somit braucht der maximale Matching Algorithmus eine Gesamtaufwand von $\mathcal{O}\left(\sqrt{n}\,(n + nnz(A))\right)$.

Ich habe den Hopcroft-Karp Algorithmus implementiert, ohne Rekursionen (beim jedem *while*-Durchlauf, um die knotendisjunkter kürzester zunehmender Pfade zu finden, ist eine Tiefensuche nötig). Dabei habe ich folgende Verbesserung eingebaut: Im obigen Algorithmus sollte man nicht mit $M = \emptyset$ anfangen. Eine erhebliche Laufzeitverbesserung wird erreicht, indem man M "naiv" initialisiert:

Algorithmus 2.4 Initialisiere "naiv" das Anfangsmatching

Beschreibung: Input: A; Output: Anfangsmatching M

1: setze $M = \emptyset$
2: **for** $(i, j) \in Struct(A)$ **do**
3: **if** i frei und j frei (bezüglich M) **then**
4: $M = M \cup \{(i, j)\}$
5: **end if**
6: **end for**

In den folgenden Tabelle sind einige Ergebnisse angegeben, die die Richtigkeit dieser Vorgehensweise untermauern. Die Testmaschine ist ein P4 HT 2.8GHz, Linux Suse 8.2. Kompiliert wurde mit der -O3 Option.

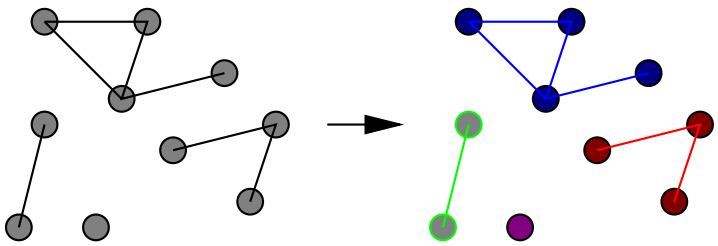
[2] wie genau wird nicht näher erläutert

Matrix Name	Dim	Laufzeit ohne Init	Laufzeit mit Init
2D_54019_highK	54019 × 54019	6.550830e-01 sec	3.256789e-02 sec
af23560	23560 × 23560	1.102160e-01 sec	2.555290e-02 sec
av41092	41092 × 41092	1.054831e+00 sec	8.022291e-01 sec
barrier2-1	113076 × 113076	3.113847e+00 sec	1.687050e-01 sec
torso1	116158 × 116158	3.188554e+00 sec	4.010559e-01 sec
ohne2	181343 × 181343	6.007443e+00 sec	3.869029e-01 sec
pre2	659033 × 659033	6.168279e+00 sec	8.084030e-01 sec
torso3	259156 × 259156	8.320268e+00 sec	2.596620e-01 sec
para-4	153226 × 153226	5.089845e+00 sec	1.926771e-01 sec
sme3Dc	42930 × 42930	1.840150e+00 sec	2.150670e-01 sec

2.2 Symmetrischer Fall: Umwandlung in blockdiagonale Gestalt

Sei $G = (V, E)$ ein Graph. Uns interessiert hier die Frage, ob der Graph in 'Inseln' partitioniert werden kann.

Die Relation: $((\forall u, w \in V)$ (u „ist verbindbar mit“ w)) $\Longleftrightarrow$ ($u = w \vee$ es gibt in G einen Pfad von u nach w) ist eine Äquivalenz in V. Seien $V_1, \ldots, V_N$ die Äquivalenzklassen (als Knotenmengen betrachtet). Diese heißen **Zusammenhangskomponenten** von G. Ist $N = 1$, so ist G zusammenhängend. Alle induzierten Teilgraphen $G\big|_{V_1}, \ldots, G\big|_{V_N}$ sind zusammenhängend und es gibt keinerlei Kanten zwischen verschiedenen Komponenten.



Ein Graph und seine Zusamenhangskomponenten

Folgender Algorithmus berechnet die Zsh-Komponenten in Zeit $\mathcal{O}(|V| + |E|)$:

Algorithmus 2.5 Finde die Zsh-Komponenten

Beschreibung: Input: G; Output: $komp\_nr$, $marker$

```
1: Sei S eine leere Schlange
2: komp_nr = 0, marker(1 : n) = 0
3: for v = 1 : n do
4:    if marker(v) == 0 then {neue Komponente entdeckt}
5:       komp_nr + +
6:       marker(v) = komp_nr
7:       insert(D, v)
8:       while D ≠ ∅ do
9:          w = remove(D)
10:         for u ∈ Adj_G(w) do
11:            if marker(u) == 0 then
12:               marker(u) = komp_nr
13:               insert(D, u)
14:            end if
15:         end for
16:      end while
17:   end if
18: end for
```

Sei jetzt A eine (strukturell) symmetrische $n \times n$ Matrix. Ist es möglich, A symmetrisch so zu permutieren, dass sie nur aus Diagonalblöcken

$$PAP^T = \begin{pmatrix} A_{1,1} & & 0 \\ & \ddots & \\ 0 & & A_{N,N} \end{pmatrix}$$

besteht? Dies hätte einen praktischen Nutzen, denn das Lösen des LGS $Ax = b$ wäre reduziert auf das Lösen von N LGSystemen mit den kleineren Matrizen $A_{1,1}, \ldots, A_{N,N}$. Es ist offensichtlich, dass dies mit den Zusammenhangskomponenten von $G(A)$ zu tun hat. Seien $V_1, \ldots, V_N$ diese Komponenten. Sei $\pi = (V_1, \ldots, V_N)$, wobei die Knotenreihenfolge innerhalb jedes $(V_k)_{k=1,\ldots,N}$ beliebig ist. Dann ist $P_\pi = I(\pi, :)$ die gesuchte Permutation. $A(\pi, \pi) = P_\pi A P_\pi^T$ besteht aus Diagonalblöcken.

2.3 Unsymmetrischer Fall: Umwandlung in Blockdreiecksgestalt

Sei $G = (V, E)$ ein Digraph. Ähnlich wie beim symmetrischen Fall ist die Relation: $((\forall u, w \in V)$ (u „ist stark verbindbar mit“ $w)) \iff (u = w \vee$ es gibt in G

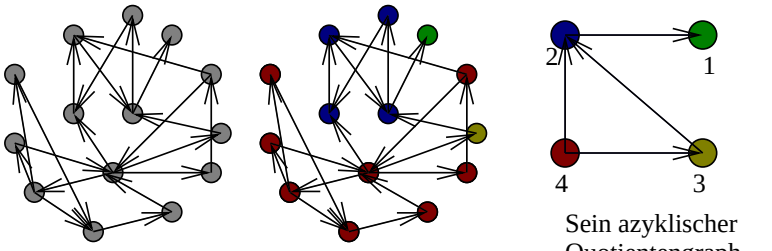
einen (gerichtete)Pfad von u nach w und einen von w nach u) eine Äquivalenz in V. Seien $V_1, \ldots, V_N$ die Äquivalenzklassen (als Knotenmengen betrachtet). Diese heißen **starke Zusammenhangkomponenten** von G. Ist $N = 1$, so ist G stark zusammenhängend. Ähnlich wie beim symmetrischen Fall, sind alle induzierten Teilgraphen $G\Big|_{V_1}, \ldots, G\Big|_{V_N}$ stark zusammenhängend. Im Unterschied zu symmetrischen Fall kann es jetzt aber Kanten zwischen verschiedenen Komponenten geben. Dafür gilt folgende elementare Eigenschaft:

Satz 2.8 *Sei $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ der **Quotientendigraph** von G bezüglich den Partition $\{V_1, \ldots, V_N\}$. Der Digraph $\mathcal{G}$ ist zyklusfrei.*

Da $\mathcal{G}$ azyklisch ist, gilt folgendes:

Korollar 2.2 *Für den Quotientendigraphen $\mathcal{G}$ gibt es eine Nummerierung seiner Knoten $V_{i_1}, V_{i_2}, \ldots, V_{i_N}$, sodass $(V_{i_\alpha}, V_{i_\beta}) \in \mathcal{E} \Longrightarrow \alpha > \beta$. (die sog. topologische Sortierung)*

Das geht, indem man immer den Knoten mit Ausgangsgrad 0 entfernt.



Sei nun A eine $n \times n$ Matrix. Ist es möglich, die Matrix A symmetrisch so zu permutieren, dass sie untere Blockdreiecksgestalt

$$PAP^T = \begin{pmatrix} A_{1,1} & & 0 \\ & \ddots & \\ A_{\widetilde{N},1} & & A_{\widetilde{N},\widetilde{N}} \end{pmatrix} \tag{2.1}$$

erhält? Sei $P \longleftrightarrow \pi$ (die zu π gehörige Permutationsmatrix) und $\pi = (\widetilde{V}_1, \ldots, \widetilde{V}_{\widetilde{N}})$ entsprechend den Diagonalblöcken $A_{1,1}, \ldots, A_{\widetilde{N},\widetilde{N}}$. Sei $\widetilde{\mathcal{G}} = (\widetilde{\mathcal{V}}, \widetilde{\mathcal{E}})$ der **Quotientendigraph** von G bezüglich dieser Partition. Folgendes elementar zu beweisendes Lemma deckt den internen Aufbau des Partitions $\widetilde{V}_1, \ldots, \widetilde{V}_{\widetilde{N}}$ auf:

Lemma 2.3 *Sei $G = (V, E)$ ein Digraph, $\widetilde{V}_1, \ldots, \widetilde{V}_{\widetilde{N}}$ eine Partition von V und $\widetilde{\mathcal{G}} = (\widetilde{\mathcal{V}}, \widetilde{\mathcal{E}})$ der **Quotientendigraph** von G bezüglich dieser Partition. Falls $\widetilde{\mathcal{G}}$ azyklisch ist, ist $\forall k \in \{1, \ldots, \widetilde{N}\}$ die Menge $\widetilde{V}_k$ die Vereinigung starker Komponenten von G.*

In unserem Fall (Gleichung 2.1) ist $\widetilde{\mathcal{G}}$ tatsächlich azyklisch, denn PAP^T ist untere (block) Dreiecksmatrix. Um also so viele Diagonalblöcke wie möglich zu haben, brauchen wir die starken Komponenten von $G(A)$. Umgekehrt: seien $V_1, \ldots, V_N$ die starken Komponenten von $G(A)$ und $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ der Quotientendigraph von G bezüglich diese Partition. Laut Korollar 2.2, gibt es eine Permutation $V_{i_1}, V_{i_2}, \ldots, V_{i_N}$, sodass $(V_{i_\alpha}, V_{i_\beta}) \in \mathcal{E} \Longrightarrow \alpha > \beta$. Sei jetzt π die Permutation $(V_{i_1}, V_{i_2}, \ldots, V_{i_N})$, wobei die Knotenreihenfolge innerhalb jedes $(V_{i_k})_{k=1,\ldots,N}$ beliebig ist. Die Matrix $P_\pi A P_\pi^T$ sieht wie im Gleichung 2.1 aus.
Nach der Transformation auf die Gestalt 2.1 ist das Lösen von LGS $Ax = b$ reduziert auf das Lösen von N kleinen LGSystemen mit Matrizen $A_{1,1}, \ldots, A_{N,N}$ und einigen (einfachen) Matrix-Vektor - Multiplikationen.

Wie im symmetrischen Fall, ist es möglich, die starken Komponenten eines Digraphen $G = (V, E)$ in Zeit $\mathcal{O}(|V| + |E|)$ zu bestimmen. Der Algorithmus dafür ([54]) ist aber bei weitem komplizierter. Es folgt eine kompakte Beschreibung dieses Algorithmus:

Definition 2.3 *Sei $G = (V, E)$ und $w \in V$.*

- $RF(G, w) = \{v |$ *es gibt in G einen Pfad von w nach v*$\}$ *(RF - Reachable From)*
- $C(w)$ *ist die starke Komponente von G welche den Knoten w enthält*

Definition 2.4 *Ein Knoten w eines Digraphen heißt* ***Wurzel*** *, falls jeder andere Knoten von w aus erreichbar ist. Ein Digraph, welcher eine Wurzel besitzt heißt* ***Wurzelgraph*** *.*

Lemma 2.4 *Jede Menge $RF(G, w)$ ist die Vereinigung einiger starker Komponenten von G. Ist $v \in RF(G, w)$, so gilt $RF(G, v) \subseteq RF(G, w)$. Falls $v \notin C(w)$, so ist $RF(G, v) \cap C(w) = \emptyset$.*

Die Knotenmengen $RF(G, w)$ bestimmt man mit einer Breiten oder Tiefensuche. Man benutzt aber im Allgemein die Tiefensuche, weil diese auch zwei Nummerierungen liefert, a - "discover first" numbers - und e - "depth first" numbers - welche wichtig sind für den Verlauf des Algorithmus. Näher erläutert wird dies bei [54]. Siehe auch Beispiel im Abbildung 2.1.

Satz 2.9 *Seien $v_1, v_2 \in RF(G, w)$. Falls $e(v_1) < e(v_2)$ mit $v_1 \notin C(v_2)$, dann $RF(G, v_1) \cap C(v_2) = \emptyset$.*

Beweis: Falls $RF(G, v_1) \cap C(v_2) \neq \emptyset$, ist $C(v_2) \subseteq RF(G, v_1)$. Das bedeutet, dass auch $v_2 \in RF(G, v_1)$. Falls $a(v_1) > a(v_2)$, d.h. v_1 ist vor v_2 "entdeckt" worden, dann müsste $e(v_1) > e(v_2)$ sein (Widerspruch!). Sei jetzt $a(v_1) < a(v_2)$, d.h. v_1 ist nach v_2 "entdeckt" worden. Aus $C(v_1) \neq C(v_2)$ ergibt sich $v_1 \notin RF(G, v_2)$. Also $e(v_1) > e(v_2)$ (Widerspruch!). □

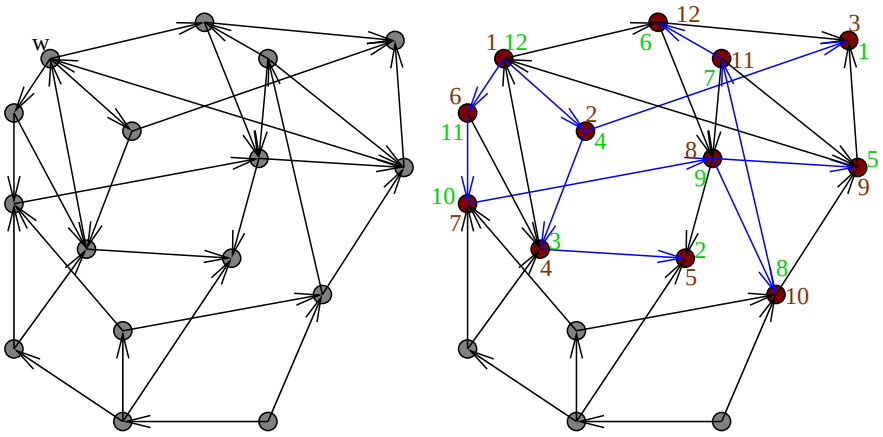


Abbildung 2.1: Beispiel für $RF(G, w)$ und die Nummern a und e

Der induzierte Teilgraph $G\big|_{RF(G,w)}$ ist ein Wurzelgraph mit Wurzel w. Man kann G in solche Wurzelgraphen in $\mathcal{O}(|V| + |E|)$ partitionieren. Seien $WG(w_1)$, ..., $WG(w_M)$ diese (knotendisjunkten) Wurzelgraphen. Die starken Komponenten eines Wurzelgraphen $D = (V_D, E_D)$ kann man in $\mathcal{O}(|E_D|)$ bestimmen:

Algorithmus 2.6 Findet die starken Zsh-Komponenten eines Wurzelgraphen D

Beschreibung: Input: $D = (V_D, E_D)$, Wurzel w; Output: $komp\_anz$, $K_{1:komp\_anz}$

1: $k = 0$
2: mit Startknoten w, bestimme die *"depth first"* Nummern e in D
3: sei $H = D^T = (V_D, E_D^T)$ der transponierte Digraph
4: **while** $V_D \neq \emptyset$ **do** {Es gibt noch unentdeckte starke Komponenten}
5: $k++$
6: Wähle r aus V_D mit $e(r)$ maximal
7: mit Startknoten r, bestimme die *"discover first"* Nummern a auf H
8: setze $K_k = \{u \in V_D | a(u) \neq 0\}$ {Die k-te Komponente}
9: setze $V_D = V_D \backslash K_k$, $H = H\big|_{V_D}$
10: **end while**
11: $komp\_anz = k$

Satz 2.10 *Im obigen Algorithmus gilt in jedem while-Schritt* $K_k = C(r)$.

Beweis: Es ist klar, dass $C(r) \subseteq K_k$. Sei $v \in K_k$ mit $v \notin C(r)$. Da $a(v) \neq 0$, gibt es einen Pfad von v nach r. Ferner $e(v) < e(r)$ und $v \notin C(r)$. Aus Satz 2.9 folgt $RF(G, v) \cap C(r) = \emptyset$, d.h. es git keinen Pfad von v nach r. Dieser Widerspruch beweist, dass $K_k = C(r)$. □

2.4 B-reduzible Normalform für strukturell nicht singuläre Matrizen

Es werden im folgenden **nur strukturell nicht singuläre Matrizen** betrachtet. Bisher wurden nur symmetrische Permutationen in Betracht gezogen, um das Lösen eines LGS in einige neue LGS mit kleineren Matrizen zu überführen. Was ist, wenn wir auch unsymmetrische Permutationen zulassen?

Definition 2.5 *Die $n \times n$ Matrix A heißt b-reduzibel (beidseitig reduzibel), falls es Permutationen $\pi \longleftrightarrow P_\pi$ und $\psi \longleftrightarrow Q_\psi$ gibt, sodass:*

$$P_\pi A Q_\psi^T = \begin{bmatrix} B_{1,1} & 0 \\ B_{2,1} & B_{2,2} \end{bmatrix} \tag{2.2}$$

Ist eine Matrix nicht b-reduzibel so heißt sie **b-irreduzibel**.

Lemma 2.5 *Eine b-irreduzible Matrix bleibt b-irreduzibel nach Anwendung von Zeilen und (oder) Spaltenpermutationen.*

Beweis: Sei B b-irreduzibel. Seien P und Q zwei beliebige Permutationen. Ist PBQ^T b-reduzibel, so gibt es $\widetilde{P}$ und $\widetilde{Q}$, sodass:

$$\widetilde{P}(PBQ^T)\widetilde{Q}^T = (\widetilde{P}P)B(Q^T\widetilde{Q}^T) = \hat{P}B\hat{Q}^T = \begin{bmatrix} B_{1,1} & 0 \\ B_{2,1} & B_{2,2} \end{bmatrix}$$

was unmöglich ist da B b-irreduzibel ist. □

Definition 2.6 *Eine b-reduzible Normalform von A ist eine Darstellung*

$$PAQ^T = \begin{bmatrix} B_{1,1} & & 0 \\ \vdots & \ddots & \\ B_{N,1} & \dots & B_{N,N} \end{bmatrix}, \quad \textit{mit } B_{i,i} \textit{ der Dimension } (n_i \times n_i)$$

wobei alle $(B_{i,i})_{i=1,\dots,N}$ b-irreduzibel[3] *sind.*

Satz 2.11 ([54]) *Sei $B = P_\pi A Q_\psi^T$ eine b-reduzible Normalform von A. Sei $\tau \longleftrightarrow T_\tau$ eine (zeilenweise) Transversale von AQ_ψ^T. Definiere $m_i = \sum_{t=1}^{i-1} n_t$. Dann gilt:*

$$\{\tau(m_i+1), \dots, \tau(m_{i+1})\} = \{\pi(m_i+1), \dots, \pi(m_{i+1})\}.$$

[3] A strukturell nicht singulär $\iff$ alle $(B_{i,i})_{i=1,\dots,N}$ strukturell nicht singulär

Beweis: Da $T_\tau A Q_\psi^T = (T_\tau P_\pi^T)B = P_{\pi^{-1}\circ\tau}B$ eine nullfreie Diagonale besitzt, muss $(\pi^{-1}\circ\tau)$ eine (zeilenweise) Transversale für B sein. Weil B (untere) Blockdreiecksmatrix ist, ist jede Transversale äquivalent mit N "Minitransversalen" für jeden Diagonalblock. Also: $\forall i = 1,\ldots,N$, $\pi^{-1}(\tau(\{m_i+1, m_i+2,\ldots,m_{i+1}\})) = \{m_i+1, m_i+2,\ldots,m_{i+1}\} \Longrightarrow \tau(\{m_i+1, m_i+2,\ldots,m_{i+1}\}) = \pi(\{m_i+1, m_i+2,\ldots,m_{i+1}\})$. Außerdem ist $[(\pi^{-1}\circ\tau)(m_i+1 : m_{i+1}) - m_i]$ eine (zeilenweise) Transversale für den Diagonalblock $B_{i,i}$ von B. □

Aus diesem Satz und Lemma 2.5 folgt:

Korollar 2.3 *Sei $B = P_\pi A Q_\psi^T$ eine b-reduzible Normalform von A. Sei $\tau \longrightarrow T_\tau$ eine (zeilenweise) Transversale von AQ_ψ^T, dann ist $T_\tau A Q_\psi^T$ eine b-reduzible Normalform mit gleichviel Diagonalblöcken. Die i-ten Blöcke haben gleiche Größe und sind Zeilenpermutierte voneinander.*

Sei jetzt PAQ^T eine b-reduzible Normalform von A und $\widetilde{T}$ eine beliebige (Zeilen) Transversale von A. Das Produkt $Q(\widetilde{T}A)Q^T$ besitzt auch eine nullfreie Diagonale. Also ist $Q\widetilde{T}$ eine Transversale für AQ^T. Aus Korollar 2.3 folgt, dass $(Q\widetilde{T})AQ^T$ eine b-reduzible Normalform mit gleichviel Diagonalblöcke ist. Die i-ten Blöcke haben gleiche Größe und sind Zeilenpermutierte voneinander. Somit sind wir bei der Umwandlung in Blockdreiecksgestalt durch symmetrische Permutationen (für $\widetilde{T}A$) gelandet. Das wurde aber im Abschnitt 2.3 behandelt.

Als **Ergebniss** halten wir fest: Sei A eine strukturell nicht singuläre Matrix und $\tau \longleftrightarrow T_\tau$ eine beliebige (zeilenweise) Transversale von ihr. Für die Matrix $T_\tau A$ sei $\psi \longleftrightarrow Q_\psi$ die Permutation aus Abschnitt 2.3. Dann ist $Q_\psi(T_\tau A)Q_\psi^T$ eine b-reduzible Normalform von A. Jede andere b-reduzible Normalform PAQ^T besitzt gleichviel Diagonalblöcke. Deren i-ten Blöcke haben gleiche Größe und sind Zeilenpermutierte voneinander.

Besitzt A eine nullfreie Diagonale, so ist also (bezüglich den Blockdreiecksgestalt) durch unsymmetrische Permutationen nicht mehr "herauszuholen" als durch symmetrische Permutationen.

Ich habe den Algorithmus zur Bestimmung der b-reduzible Normalform (BRNF) implementiert. Alle Algorithmen die hier erwähnt wurden, habe ich ohne jegliche Rekursion[4] implementiert. Die Testmaschine war ein P4 HT 2.8GHz, Linux Suse 8.2. Kompiliert wurde mit den -O3 Option. Die vergleichbare Matlab 6.1 built-in Funktion $[p, q, r, s] = dmperm(A)$ ist abgestürzt für alle hier aufgeführten Testmatrizen, mit dem Fehlermeldung *Speicherzugriffsfehler*.

[4] wegen die Tiefensuche

Matrix Name	Dim	BRNF Laufzeit	st. Komp.
2D_54019_highK	54019×54019	1.101671e-01 sec	7157
av41092	41092×41092	1.097216e+00 sec	4
barrier2-1	113076×113076	6.525649e-01 sec	1
Hamrle3	1447360×1447360	6.719517e+01 sec	1
para-6	155924×155924	5.865860e-01 sec	565
pre2	659033×659033	1.896706e+00 sec	29282
torso3	259156×259156	8.742669e-01 sec	1
stomach	213360×213360	6.006340e-01 sec	1
venkat50	62424×62424	3.077969e-01 sec	1

2.5 Gewichtete Transversalen

Eine weitere Verbesserung bei der Stabilität der Gauß Elimination ist in den letzten Jahren durch die Anwendung von gewichteten Transversalen erzielt worden. Die Idee dahinter macht auch Sinn: Wenn es mehrere Transversalen gibt, dann suche diejenigen heraus, welche gut für die Gauß-Stabilität sind. Die unten beschriebenen Herangehensweisen sind zwar von heuristischer Natur, in der Praxis aber arbeiten sie gut. Es wird versucht, möglichst große Einträge auf die Diagonale zu permutieren.

2.5.1 Die Flaschenhals-Transversale

Sei A eine (wenigstens strukturell) nicht singuläre $n \times n$ Matrix. Gesucht wird eine Transversale T (in Matchingform), so dass für jede andere Transversale T_1 gilt:

$$\min_{(i,j)\in T_1} |A_{i,j}| \leq \min_{(i,j)\in T} |A_{i,j}|$$

In diesem Fall heißt T eine **Flaschenhals-Transversale** und das kleinste $(|A_{i,j}|)_{(i,j)\in T}$ heißt der Flaschenhals-Wert.

Der BT[5] Algorithmus aus [31] benötigt als Komponente einen Algorithmus, der ein maximales Matching (im bipartiten Graph $BG(A)$ von A) berechnet. Sei $MT(A, M)$ eine Routine, welche für eine Matrix A und ein Anfangsmatching M ein maximales Matching liefert (alle Matchings in $BG(A)$). Für $\epsilon \geq 0$ definieren wir die Matrix A_ϵ wie folgt:

$$A_\epsilon(i,j) = \begin{cases} A(i,j) & \text{falls } |A(i,j)| \geq \epsilon \\ 0 & \text{sonst} \end{cases}.$$

[5] BT wie Bottleneck Transversal

Wir ersetzen also alle Einträge $|A(i,j)| < \epsilon$ durch 0. Zum Beispiel ist $A_0 = A$, $A_{+\infty} = 0$. Entfernt man aus einem gegebene Matching M alle Kanten $(i,j) \in M$ mit $|A(i,j)| < \epsilon$, so bezeichnen wir das resultierende Matching mit M_ϵ. Zum Beispiel ist $M_0 = M$, $M_{+\infty} = \emptyset$.

Algorithmus 2.7 BT : findet eine flaschenhals Transversale

Beschreibung: Input: A strukturell nicht singulär; Output: M

1: $\epsilon_{min} = 0$, $\epsilon_{max} = +\infty$
2: $M = MT(A, \emptyset)$
3: **while** es gibt i, j mit $\epsilon_{min} < |A_{i,j}| < \epsilon_{max}$ **do**
4: $\quad \epsilon = |A_{i,j}|$
5: $\quad M' = MT(A_\epsilon, M_\epsilon)$ $\{M_\epsilon$ damit man nicht mit $\emptyset$ anfängt$\}$
6: $\quad$ **if** M' perfekt **then**
7: $\quad\quad M = M'$
8: $\quad\quad \epsilon_{min} = \epsilon$
9: $\quad$ **else**
10: $\quad\quad \epsilon_{max} = \epsilon$
11: $\quad$ **end if**
12: **end while**

Die Werte ϵ_{min} und ϵ_{max} sind so gewählt, dass immer $A_{\epsilon_{min}}$ ein perfektes Matching besitzt und $A_{\epsilon_{max}}$ kein perfektes Matching besitzt. Am Ende des Algorithmus wird $\epsilon_{min} = |A(i',j')|$ gelten und für alle $\epsilon > |A(i',j')|$, besitzt A_ϵ kein perfektes Matching. Dieses $|A(i',j')|$ wird gleichzeitig der Flaschenhals-Wert sein.

Unter Verwendung des Hopcroft-Karp Algorithmus zur Bestimmung von M', kann der obige Algorithmus mit den Aufwand $\mathcal{O}(\sqrt{n}\, nnz(A)\, \log_2(nnz(A)))$ implementiert werden. Diese Obergrenze ist aber sehr pessimistisch. Bei der Aufwandsanalyse wird überhaupt nicht berücksichtigt, dass die Bestimmung von M' (Zeile 5) mit einem Anfangsmatching M_ϵ anfängt.

2.5.2 Die MPD Transversale

Die MPD Transversale ([32]) ist die bisher beste Transversale bezüglich der Stabilität des Gauß Algorithmus. Ich werde sie hier eingehend erläutern.

Sei A eine (wenigstens strukturell) nicht singuläre Matrix. Gesucht wird eine Transversale T (in Matchingform), so dass für jede andere Transversale T_1 gilt:

$$\prod_{(i,j)\in T_1} |A_{i,j}| \leq \prod_{(i,j)\in T} |A_{i,j}| \tag{2.3}$$

Daher auch der Name "Maximum Product on Diagonal Transversale", kurz MPD.

Sei $A = (a_{i,j})_{i,j=1,\ldots,n}$ und für jeder Zeile $i = 1, \ldots, n$ sei $a_i = \max_{j=1,\ldots,n} |a_{i,j}|$. Definiere:

$$C = (c_{i,j})_{i,j=1,\ldots,n} = \begin{cases} \log \frac{a_i}{|a_{i,j}|} & \text{falls } a_{i,j} \neq 0 \\ +\infty & \text{falls } a_{i,j} = 0 \end{cases} \tag{2.4}$$

Es gilt $C \geq 0$. Für jede (Spalten) Transversale ψ gilt

$$\log \frac{\prod_{i=1}^{n} a_i}{\prod_{i=1}^{n} |a_{i,\psi_i}|} = \sum_{i=1}^{n} \log \frac{a_i}{|a_{i,\psi_i}|} = \sum_{i=1}^{n} c_{i,\psi_i}. \tag{2.5}$$

Daraus folgt:

Satz 2.12 *Eine MPD Transversale für A ist äquivalent zu einer minimal-gewichteten Transversalen (kurz MWT) für C.*

Wir können uns also auf das *MWT* Problem für nicht negative Matrizen C konzentrieren. Der Algorithmus dafür braucht als Komponente einen Algorithmus, der in einem gewichteten Digraphen den Abstand und den dazugehörigen gerichteten Pfad mit minimalem Gewicht eines Knotens zu einer (Knoten) Menge bestimmt. Das geht durch einige leichte Modifikationen von Dijkstras Algorithmus.

2.5.2.1 Dijkstras Algorithmus

Sei $G = (V, E, g)$ ein gewichteter Digraph, wobei die Gewichtfunktion $g : E \to \mathbb{R}^+$ nichtnegativ ist.

Den **Abstand** $\delta_g(x, y)$ von $x \in V$ nach $y \in V$ definiert man als:

$$\delta_g(x,y) = \begin{cases} +\infty & \left\{ \begin{array}{c} \text{falls kein gerichteter Pfad von } x \text{ nach } y \\ \text{existiert, d.h. y ist von } x \text{ aus unerreichbar} \end{array} \right\} \\ \min(\sum_{j=1}^{t} g(w_{j-1}w_j)) & \left\{ \begin{array}{c} \text{für alle gerichtete Pfade} \\ x = w_0, w_1, \ldots, w_t = y \text{ von } x \text{ nach } y \end{array} \right\} \end{cases}$$

Seien $V_1, V_2 \subseteq V$. Der Abstand von V_1 zu V_2 ist definiert durch:

$$\delta_g(V_1, V_2) = \min\{\delta_g(x, y) \mid x \in V_1, y \in V_2\}$$

Satz 2.13 $\forall x, y, z \in V$ *gilt* $\delta_g(x, y) \leq \delta_g(x, z) + \delta_g(z, y)$.

Beweis: Sei zuerst y von x aus unerreichbar. Falls z von x aus erreichbar ist und y von z aus erreichbar ist, dann müsste auch y von x aus erreichbar sein (Widerspruch!). Also, ist $\delta_g(x,y) = +\infty$, so muss $\delta_g(x,z) = +\infty$ oder $\delta_g(z,y) = +\infty$ gelten. Sei nun y von x aus erreichbar. Falls z von x aus unerreichbar ist oder y von z aus unerreichbar ist, dann stimmt unsere Ungleichung. Zuletzt, sei z von x aus erreichbar und y von z aus erreichbar. Es gibt einen Pfad mit Gewicht $\delta_g(x,z)$, welcher x mit z verbindet und ein Pfad mit Gewicht $\delta_g(z,y)$, welcher z mit y verbindet. Dadurch ergibt sich ein Pfad von x nach y, mit den Gewicht $\delta_g(x,z) + \delta_g(z,y)$. Jetzt ist unsere Ungleichung klar. □

Satz 2.14 *Sei $x = w_0, w_1, \ldots, w_t = y$ ein (gerichteter) Pfad mit minimalem Gewicht, d.h. $\delta_g(x,y) = \sum_{j=1}^{t} g(w_{j-1}w_j)$. Dann besitzt für jedes $k < j$ aus $\{0,\ldots,t\}$, auch der (gerichteter) Pfad $w_k, w_{k+1}, \ldots, w_j$ minimales Gewicht unter allen Pfaden von w_k nach w_j.*

Für ein festes $v \in V$, berechnet Dijkstras Algorithmus alle Abstände $(\delta_g(v,w))_{w\in V}$:

Algorithmus 2.8 Dijkstras Algorithmus (leicht modifiziertes Version)

Beschreibung: Input: $G = (V,E,g)$ mit $g \geq 0$; Output: $(\delta_g(v,w))_{w\in V}$

```
 1: seien Q und B zwei leere Datastrukturen
 2: ∀w ∈ V setze d(w) = +∞
 3: setze d(v) = 0 und füge v in Q ein
 4: while Q ≠ ∅ do
 5:    entferne w aus Q mit d(w) minimal
 6:    füge w in B ein
 7:    for (w,z) ∈ E mit z ∉ B do
 8:       if z ∉ Q then {bis jetzt war z unentdeckt und es galt d(z) = +∞}
 9:          d(z) = d(w) + g(w,z)
10:          füge z in Q ein
11:       else if d(z) > d(w) + g(w,z) then {z ist schon entdeckt, d(z) < +∞}
12:          d(z) = d(w) + g(w,z)
13:       end if
14:    end for
15: end while
```

In Algorithmus 2.8, enthält die Menge Q Knoten w, für die es gesichert ist, dass es einen (gerichteten) Pfad von v nach w gibt. Sobald der Abstand von v nach w feststeht, wird w nach B verschoben.

Satz 2.15 ([21]) *Sei $B = (\underbrace{v_0}_{v}, v_1, \ldots, v_k)$ die Folge der Knoten w, die aus Q*

entfernt werden (Zeilen 5,6). Für alle $i = 0, 1, \ldots, k$ *gilt* $d(v_i) = \delta_g(v, v_i)$ *und für alle* $w \notin \{v_0, v_1, \ldots, v_k\}$ *gilt* $d(w) = \delta_g(v, w) = +\infty$ *(d.h.* w *ist von* v *aus unerreichbar). Darüber hinaus gilt* $0 = d(v_0) \leq d(v_1) \leq \ldots \leq d(v_k)$ $(< +\infty)$.

Verwendet man für D einen Heap, ist der obige Algorithmus in

$$\mathcal{O}((|V| + |E|) \log |V|)$$

implementierbar.

Sei $\widetilde{V} \subset V$. Da $0 = d(v_0) \leq d(v_1) \leq \ldots \leq d(v_k)$ $(< +\infty)$ gilt, kann mann mit Dijkstras Algorithmus auch den Abstand $\delta_g(v, \widetilde{V})$ bestimmen:

Algorithmus 2.9 Bestimme den Abstand $\delta_g(v, \widetilde{V})$

Beschreibung: Input: $G = (V, E, g)$ mit $g \geq 0$, $\widetilde{V} \subset V$; Output: $\delta_g(v, \widetilde{V})$

1: seien Q und B zwei leere Datastrukturen
2: $\forall w \in V$ setze $d(w) = +\infty$
3: setze $d(v) = 0$ und füge v in Q ein
4: **while** $Q \neq \emptyset$ **do**
5: entferne w aus Q mit $d(w)$ minimal
6: **if** $w \in \widetilde{V}$ **then** {minimale Abstand gefunden}
7: **return** $d(w)$
8: **end if**
9: {jetzt gilt $w \notin \widetilde{V}$}
10: füge w in B ein
11: **for** $(w, z) \in E$ mit $z \notin B$ **do**
12: **if** $z \notin Q$ **then** {bis jetzt war z unentdeckt und es galt $d(z) = +\infty$}
13: $d(z) = d(w) + g(w, z)$
14: füge z in Q ein
15: **else if** $d(z) > d(w) + g(w, z)$ **then** {z ist schon entdeckt, $d(z) < +\infty$}
16: $d(z) = d(w) + g(w, z)$
17: **end if**
18: **end for**
19: **end while**
20: **return** $(+\infty)$ {$\widetilde{V}$ ist von v aus nicht erreichbar}

Betrachtet man die (gerichteten) Pfade mit minimalem Gewicht von v nach allen $w \in V$, so erhält man einen Wurzelbaum mit Wurzel v (unerreichbare Knoten werden nicht betrachtet). Deswegen kann der obige Algorithmus mit einen Array *Vorg* (Vorgänger) nachgerüstet werden, um auch den gerichteten Pfad mit minimalem Gewicht von v nach $\widetilde{V}$ zu bestimmen.[6]

[6] Falls $w \in \widetilde{V}$ mit $\delta(v, w) = \delta_g(v, \widetilde{V})$, dann ist der (eigentlich umgekehrte) Pfad mit minimalem Gewicht: $w, Vorg(w), Vorg(Vorg(w)), \underbrace{\ldots}_{\text{while } Vorg(\ldots Vorg(w)\ldots)\neq -1}, \underbrace{Vorg(\ldots Vorg(w)\ldots)}_{v}$

Algorithmus 2.10 Bestimme den Abstand $\delta_g(v, \widetilde{V})$ und den Pfad mit minimalen Gewicht dazu

Beschreibung: Input: $G = (V, E, g)$ mit $g \geq 0$, $\widetilde{V} \subset V$; Output: $\delta_g(v, \widetilde{V})$, $w \in \widetilde{V}$ mit $\delta(v, w) = \delta_g(v, \widetilde{V})$ und $Vorg$

```
1: seien Q und B zwei leere Datastrukturen
2: ∀w ∈ V setze d(w) = +∞, Vorg(w) = 0
3: setze d(v) = 0 und füge v in Q ein
4: Vorg(v) = −1 {erkläre v als root}
5: for (v, w) ∈ E do
6:    setze Vorg(w) = v
7: end for
8: while Q ≠ ∅ do
9:    entferne w aus Q mit d(w) minimal
10:   if w ∈ Ṽ then {minimale Abstand gefunden}
11:      return d(w), w, Vorg
12:   end if
13:   {jetzt gilt w ∉ Ṽ}
14:   füge w in B ein
15:   for (w, z) ∈ E mit z ∉ B do
16:      if z ∉ Q then {bis jetzt war z unentdeckt und galt d(z) = +∞}
17:         d(z) = d(w) + g(w, z)
18:         Vorg(z) = w
19:         füge z in Q ein
20:      else if d(z) > d(w) + g(w, z) then {z ist schon entdeckt, d(z) < +∞}
21:         d(z) = d(w) + g(w, z)
22:         Vorg(z) = w
23:      end if
24:   end for
25: end while
26: return (+∞) {Ṽ ist von v aus nicht erreichbar}
```

Es ist naheliegend, für Q einen Heap zu verwenden.

2.5.2.2 Minimum Weight Transversale

Sei $G = (V_r, V_c, E, c)$ der gewichtete bipartite Graph der Matrix C aus Satz 2.12. Es gilt $|V_r| = |V_c| = n$ und $\forall e \in E$, $c(e) \geq 0$. Dabei ist $c(e) = c_{i,j}$ das Gewicht der Kante $e = (i, j) \in E$. Die Paare (i, j) mit $c_{i,j} = +\infty$ werden nicht als Kanten betrachtet, denn $A_{i,j} \neq 0 \Longleftrightarrow c_{i,j} < +\infty$ und solche Paare (i, j) mit $A_{i,j} = 0$ sind für unseren Problem sowieso völlig irrelevant.

Gesucht wird ein **perfektes** Matching $\widetilde{M} \subset E$, so dass dessen Gewicht:

$$c(\widetilde{M}) = \sum_{(i,j)\in\widetilde{M}} c_{i,j} \tag{2.6}$$

minimal ist.

Sei M ein nicht perfektes Matching und p ein zunehmender Pfad bezüglich M. Wir definieren die **Pseudolänge** des zunehmendes Pfades p als:

$$l(p) = c(M \triangle p) - c(M) = c(p \backslash M) - c(M \cap p)$$

Definition 2.7 *Ein Matching M heißt **extrem**, falls es zwei Vektoren $u, v \in \mathbb{R}^n$ $(\infty \notin \mathbb{R})$ gibt mit:*

$$\forall (i,j) \in E,\ u_i + v_j \leq c_{i,j} \tag{2.7}$$
$$\forall (i,j) \in M,\ u_i + v_j = c_{i,j} \tag{2.8}$$

Eigenschaft 2.7 gilt auch für (i,j) mit $c_{i,j} = +\infty$.

Satz 2.16 *Falls ein perfektes Matching M extrem ist, dann besitzt dieses Matching minimales Gewicht.*

Beweis: Sei $\widetilde{M}$ eine perfektes extremes Matching. Dann gilt:

$$c(\widetilde{M}) = \sum_{(i,j)\in\widetilde{M}} c_{i,j} \overbrace{=}^{(2.8)} \sum_{(i,j)\in\widetilde{M}} (u_i + v_j) \overbrace{=}^{\widetilde{M} \text{ perfekt}} \sum_{i=1}^{n} u_i + \sum_{j=1}^{n} v_j \ .$$

Für ein beliebiges perfektes Matching M gilt:

$$c(M) = \sum_{(i,j)\in M} c_{i,j} \overbrace{\geq}^{(2.7)} \sum_{(i,j)\in M} (u_i + v_j) \overbrace{=}^{M \text{ perfekt}} \sum_{i=1}^{n} u_i + \sum_{j=1}^{n} v_j = c(\widetilde{M})$$

□

Nimmt man $u = v = (0, \ldots, 0)^T$, so ist (2.7) immer erfüllt. In einem leeren Matching $M = \emptyset$ gibt es keine Kante, die die Gleichungen (2.8) nicht erfüllt. Also, ein leere Matching ist extrem. Wie es weiter geht, zeigt der folgender Satz:

Satz 2.17 *Sei M ein extremes aber nicht perfektes Matching. Für jeden freien Startknoten $j_0 \in V_c$ (bezüglich M), gibt es einen zunehmenden Pfad $p : j_0, \ldots, i_0$ bezüglich M, wobei i_0 notwendig frei in V_r ist, sodass das Matching $M' = M \triangle p$ extrem ist.*

Der Beweis dieses Satzes ist konstruktiv. Er wird uns den zunehmenden Pfad liefern und die Aufdatierungen für u und v.

Beweis: Seien u und v die Vektoren der Definition 2.7. Definiere $(\overline{c}_{i,j})$ nach:

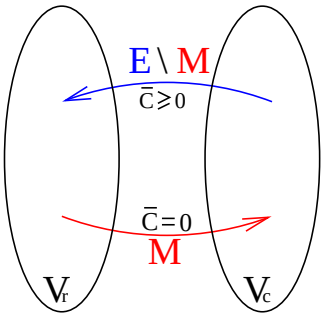
$$\overline{c}_{i,j} = c_{i,j} - u_i - v_j$$

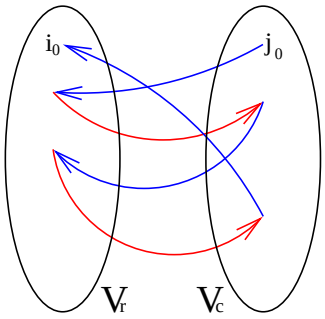
Dies sind die sogenannten **reduzierten Gewichte**. Es gilt $\forall (i,j) \in E$, $\overline{c}_{i,j} \geq 0$ (aus 2.7) und $\forall (i,j) \in M$, $\overline{c}_{i,j} = 0$ (aus 2.8). Da M nicht perfekt ist, gibt es (mindestens) einen zunehmenden Pfad, der bei einem freien Knoten j_0 startet. Zwischen alle diese wähle dasjenige p mit minimaler reduzierter Pseudolänge:

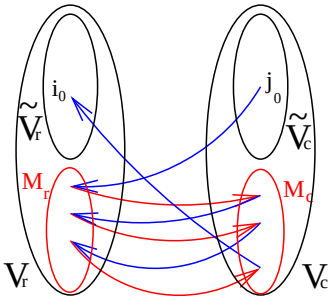
$$\overline{l}(p) = \overline{c}(p \backslash M) - \overline{c}(M \cap p) = \overline{c}(p \backslash M) + \overline{c}(M \cap p) = \overline{c}(p)$$

p ist die sog. *sap*, shortest augmenting path. Hier ist von entscheidender Bedeutung, dass das reduzierte Gewicht aller Kanten aus M null ist. Betrachten wir nun den gewichteten bipartiten Graphen $(V_r, V_c, E, \overline{c})$ und die Kanten aus E als gerichtet wie folgt:

$$\text{ist } (i,j) \in E \backslash M \Longrightarrow \text{Richtung } i \longleftarrow j$$
$$\text{ist } (i,j) \in M \Longrightarrow \text{Richtung } i \longrightarrow j$$







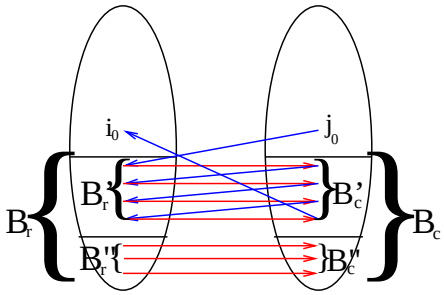
sap

Seien $\widetilde{V}_r$, $\widetilde{V}_c$ die Mengen aller freien Knoten aus V_r bzw. V_c, und M_r, M_c die Mengen aller saturierten Knoten aus V_r bzw. V_c. Die Suche nach *sap* ist nichts anders als die Suche in $(V_r, V_c, E, \overline{c})$ nach dem Abstand $\delta_{\overline{c}}(j_0, \widetilde{V}_r)$ von $j_0 \in \widetilde{V}_c$ nach $\widetilde{V}_r$ zusammen mit dem Pfad, bei dem dieser minimale Abstand erreicht wird.

Da die Gewichtsfunktion nichtnegativ ist, können wir die letzte Variante des Dijkstras Algorithmus (Seite 34) ins Spiel bringen.

Hier ist als Startknoten $v = j_0$. Seien B, Q, $i_0 = w$ und $sap : j_0, \ldots, i_0$ dessen Output. Entfernen wir j_0 aus B: $B = B \backslash \{j_0\}$. Jetzt ist klar, dass $B \subset M_r \cup M_c$.

Falls ein Knoten $i \in M_r$ in B eingefügt wird, dann gibt es nur eine Möglichkeit von ihm weg zu gehen, nach $j \in M_c$ mit $(i,j) \in M$. Da $\overline{c}_{i,j} = 0$, können wir annehmen, dass j der nächste Knoten ist, welche in B eingefügt wird[7]. Also kann B in $B_r = B \cap M_r$ und $B_c = B \cap M_c$ partitioniert werden, wobei B_r und B_c bijektiv durch M miteinander verbunden sind. *sap* fängt bei j_0 an, geht dann alternierend in B_r und B_c und endet in $i_0 \in \widetilde{V}_r$ (siehe Abb. *sap*).



Ihrerseits werden B_r und B_c in zwei Teile partitioniert durch *sap*: B'_r, B'_c die Knoten die zu *sap* gehören und B''_r und B''_c die Knoten die nicht zu *sap* gehören. Auch B'_r und B'_c sowie B''_r und B''_c sind bijektiv durch M mit einander verbunden.
Sei $lsap = \delta_{\overline{c}}(j_0, \widetilde{V}_r) = d(i_0)$. Bekanntlich, gilt $(\forall w \in B = B_r \cup B_c)\ d(i_0) \geq d(w)$. Da die Indizes $k = 1, \ldots, n$ sowohl in V_r als auch in V_c auftreten, werden wir $d_{col}(k)$ benutzen, dort wo es Missverständnise geben kann, um zu unterscheiden, dass es sich um eine Spalte handelt, d.h. $k \in V_c$. Andererseits, falls $i \in B_r$ und $(i,j) \in M$ $(\Rightarrow j \in B_c)$, dann $d_{col}(j) = d(i)$. Es ist auch bekannt, dass für $w_1, w_2 \in B$, falls w_1 früher als w_2 in B eingefügt wurde, dann $\underbrace{d(w_1)}_{\delta_{\overline{c}}(i_0,w_1)} \leq \underbrace{d(w_2)}_{\delta_{\overline{c}}(i_0,w_2)}$.

Als zunehmender Pfad p weden wir *sap* benutzen. Um zu zeigen, dass das nächste Matching $M' = M \triangle p$ extrem ist, müssen wir die Vektoren u und v so modifizieren, dass die Eigenschaften 2.7 und 2.8, jetzt auf M' bezogen, Geltung haben. Modifizieren wir u und v wie folgt:

$$\forall i \in B_r,\ u'_i = u_i + d(i) - lsap \tag{2.9}$$

$$\forall (i,j) \in M',\ v'_j = c_{i,j} - u'_i \tag{2.10}$$

Aus 2.10 sieht man sofort, dass die Eigenschaft 2.8 (bezüglich M', u' und v') erfüllt ist. Es bleibt zu beweisen, dass die neuen reduzierten Gewichte $\overline{c}'_{i,j}$ nichtnegativ sind:

$$\overline{c}'_{i,j} = c_{i,j} - u'_i - v'_j \geq 0,\ \forall (i,j) \in E \tag{2.11}$$

Sei $\triangle u_i = u'_i - u_i$, $\triangle v_j = v'_j - v_j$, $\triangle \overline{c}_{i,j} = \overline{c}'_{i,j} - \overline{c}_{i,j}$.

- $\forall i \in V_r \backslash (B_r \cup \{i_0\})$, $\triangle u_i = 0$ d.h. u ändert sich nur in $B_r \cup \{i_0\}$[8]
- $\forall j \in V_c \backslash (B_c \cup \{j_0\})$, $\triangle v_j = 0$ d.h. v ändert sich nur in $B_c \cup \{j_0\}$

$\forall i \in (B_r \cup \{i_0\})$, $\triangle u_i = d(i) - lsap = d(i) - d(i_0) \leq 0$.
$\forall j \in (B_c \cup \{j_0\})$, sei $(i,j) \in M'$, $\triangle v_j = c_{i,j} - u'_i - v_j = (\overline{c}_{i,j} + u_i + v_j) - u'_i - v_j = \overline{c}_{i,j} - \triangle u_i$.

[7] es bedarf nur eine kleine Veränderung beim Dijkstras Algorithmus
[8] eigentlich $\triangle u_{i_0} = 0$, d.h. u ändert sich nur in B_r

$\forall(i,j) \in E$, $\overline{c}'_{i,j} = c_{i,j} - u'_i - v'_j = (c_{i,j} - u_i - v_j) - ((u'_i - u_i) + (v'_j - v_j)) = \overline{c}_{i,j} - (\triangle u_i + \triangle v_j)$, d.h. 2.11 ist äquivalent zu:

$$(\forall(i,j) \in E)(\triangle u_i + \triangle v_j \leq \overline{c}_{i,j}) \tag{2.12}$$

Sei nun $(i,j) \in E$ beliebig. Es gibt vier mögliche Fälle:

1. $i \notin B_r \cup \{i_0\}$, $j \notin B_c \cup \{j_0\}$. Da $\triangle u_i = \triangle v_j = 0$ und $0 \leq \overline{c}_{i,j}$, ist 2.12 bewiesen.

2. $i \in B_r \cup \{i_0\}$, $j \notin B_c \cup \{j_0\}$. Da $\triangle u_i \leq 0$, $\triangle v_j = 0$ und $0 \leq \overline{c}_{i,j}$, ist 2.12 bewiesen.

3. $i \notin B_r \cup \{i_0\}$, $j \in B_c \cup \{j_0\}$. $\triangle u_i = 0$. Hier unterscheidet man 2 Fälle:

   (a) $j \in B''_c$. Sei $k \in B''_r$ mit $(k,j) \in M'$. Da auch $(k,j) \in M$, $\overline{c}_{k,j} = 0$. $\triangle v_j = \overline{c}_{k,j} - \triangle u_k = -\triangle u_k = -d(k) + d(i_0) = -d_{col}(j) + d(i_0)$. Da $i \notin B_r \cup \{i_0\}$, gilt $d(i_0) = \delta_{\overline{c}}(j_0, i_0) \leq \delta_{\overline{c}}(j_0, i) \underbrace{\leq}_{\text{Satz 2.13}} \underbrace{\delta_{\overline{c}}(j_0, j)}_{d_{col}(j)} + \delta_{\overline{c}}(j, i) \leq d_{col}(j) + \overline{c}_{i,j}$. Also $-d_{col}(j) + d(i_0) \leq \overline{c}_{i,j}$ und 2.12 ist somit bewiesen.

   (b) $j \in B'_c \cup \{j_0\}$. Sei $k \in B'_r$ und $k' \in B'_r \cup \{i_0\}$ sodass $(k,j) \in M$ und $(k',j) \in M'$. Dann sieht *sap* so aus: $j_0, \ldots, k, j, k', \ldots, i_0$. Da *sap* eine (gerichtete)Pfad mit minimale Gewicht ist, folgt aus 2.14 dass $d(k') = \delta_{\overline{c}}(j_0, k') = d_{col}(j) + \overline{c}_{k',j}$. $\triangle v_j = \overline{c}_{k',j} - \triangle u_{k'} = \overline{c}_{k',j} - (d(k') - d(i_0)) = d(i_0) - d_{col}(j)$. Dass $d(i_0) - d_{col}(j) \leq \overline{c}_{i,j}$ stimmt, wird genau wie oben gezeigt.

4. $i \in B_r \cup \{i_0\}$, $j \in B_c \cup \{j_0\}$. $\triangle u_i = d(i) - d(i_0)$. Sei $k \in B_r \cup \{i_0\}$ sodass $(k,j) \in M'$. Dann $\triangle v_j = \overline{c}_{k,j} - \triangle u_k = \overline{c}_{k,j} - (d(k) - d(i_0))$. Z.z $\triangle u_i + \triangle v_j \leq \overline{c}_{i,j} \Leftrightarrow (d(i) - d(i_0)) + \overline{c}_{k,j} - (d(k) - d(i_0)) \leq \overline{c}_{i,j} \Leftrightarrow d(i) \leq (d(k) - \overline{c}_{k,j}) + \overline{c}_{i,j}$. Hier unterscheidet man auch zwei Fälle:

   (a) $j \in B''_c$. Dann $k \in B''_r$ und $(k,j) \in M$. Deswegen $\overline{c}_{k,j} = 0$. Z.z. $d(i) \leq d(k) + \overline{c}_{i,j}$. Es gilt $d(k) = d_{col}(j)$ und aus Satz 2.13 folgt $d(i) = \delta_{\overline{c}}(j_0, i) \leq \underbrace{\delta_{\overline{c}}(j_0, j)}_{d_{col}(j)} + \delta_{\overline{c}}(j, i) \leq d(k) + \overline{c}_{i,j}$.

   (b) $j \in B'_c \cup \{j_0\}$. Die *sap* sieht so aus: $j_0, \ldots, j, k, \ldots, i_0$. Wie oben, $d(k) - \overline{c}_{k,j} = d_{col}(j)$. Z.z. $d(i) \leq d_{col}(j) + \overline{c}_{i,j}$. Das folgt genau wie eben gezeigt wurde.

Damit ist der Satz bewiesen. □

Im obigen Satz ist die Menge B_c eher eine dummy Menge denn für ein $i \in M_r$ gibt es genau eine Möglichkeit nach V_c überzuspringen, nämlich $i \to j$ wobei

$(i, j) \in M$. So lassen sich einige überflüssige Variablen umgehen. Jetzt können wir den Dijkstras Algorithmus an unseren Gegebenheiten anpassen, um den zunehmenden Pfad *sap*, die Menge $B = B_r$, die Abstände d und $lsap$ zu bestimmen:

Algorithmus 2.11 sap

Beschreibung: Input: Bipartiter gewichteter Graph $(V_r, V_c, E, \overline{c})$ mit $\overline{c} \geq 0$, Matching M, $j_0 \in V_c$ frei bezüglich M; Output: B, d, i_0, $Vorg$ und $lsap$

1: Seien Q und B zwei leere Datastrukturen {werden nur Elemente aus V_r enthalten}
2: $\forall w \in V_c$ setze $d(w) = +\infty$, $Vorg(w) = 0$
3: **for** $(i, j_0) \in E$ **do** {Navigiere in die Spalte j_0}
4: $d(i) = \overline{c}_{i,j_0}$ und füge i in Q ein
5: $Vorg(i) = -1$ {$Vorg(i) = -1 \Leftrightarrow$ direkte Söhne von j_0}
6: **end for**
7: **while** `true` **do**
8: **if** $Q == \emptyset$ **then**
9: **return** `Error: Matrix ist strukturell singulär`
10: **end if**
11: entferne akt_row aus Q mit $d(akt\_row)$ minimal
12: **if** akt_row frei bezüglich M **then** {*sap* gefunden}
13: **return** B, $i_0 = akt\_row$, d, $Vorg$, $lsap = d(akt\_row)$
14: **end if**
15: füge akt_row in B ein
16: sei $j \in V_c$ mit $(akt\_row, j) \in M$
17: **for** $(i, j) \in E$ mit $i \notin B$ **do** {Navigiere in die Spalte j}
18: **if** $i \in Q$ **then** {i ist schon entdeckt, $d(i) < +\infty$}
19: **if** $d(i) > d(akt\_row) + \overline{c}_{i,j}$ **then** {Verbesserung möglich}
20: $d(i) = d(akt\_row) + \overline{c}_{i,j}$
21: $Vorg(i) = akt\_row$
22: **end if**
23: **else** {$i \notin Q$ tritt zum ersten Mal auf, $d(i) == +\infty$}
24: $d(i) = d(akt\_row) + \overline{c}_{i,j}$
25: $Vorg(i) = akt\_row$
26: füge i in Q ein
27: **end if**
28: **end for**
29: **end while**

Hat man den Vektor $Vorg$, dann baut man den Pfad *sap* wie folgt:

Für einen saturierten Knoten k, bezeichnen wir $M(k)$ den eindeutigen Knoten

k', sodass $(k,k') \in M$ (oder $(k',k) \in M$). Das umgekehrte *sap* ist dann:

$$i_0, M(Vorg(i_0)), Vorg(i_0), M(Vorg(Vorg(i_0))), Vorg(Vorg(i_0)),$$

$$\underbrace{\ldots}_{\text{while } Vorg(\ldots Vorg(i_0)\ldots)\neq -1}, M(Vorg(\ldots Vorg(i_0)\ldots)), Vorg(\ldots Vorg(i_0)\ldots), j_0$$

Damit ist der Hauptschritt vollständig behandelt worden.
Der *MWT* Algorithmus[9] sieht dann so aus:

Algorithmus 2.12 *MWT* Algorithmus

Beschreibung: Input: Bipartiter gewichteter Graph (V_r, V_c, E, c), ein extremes Anfangsmatching M und die dazugehörigen Vektoren u,v (siehe Definition 2.7); Output: *MWT*-Transversale M

1: **for** $j_0 = 1, \ldots, n$ **do**
2: **if** j_0 frei bezüglich M **then**
3: berechne die reduzierten Gewichte: $(\forall (i,j) \in E)\, \overline{c}_{i,j} = c_{i,j} - u_i - v_j$
4: $[d, sap, B, lsap] = sap(V_r, V_c, E, \overline{c}, M, j_0)$ {Hauptschritt}
5: datiere M auf: $M = M \triangle sap$
6: datiere u auf: $(\forall i \in B)\, u_i = u_i + d(i) - lsap$
7: datiere v auf: $(\forall j \in V_c)$ saturiert, $v_j = c_{i,j} - u_i$ wobei $(i,j) \in M$
8: **end if**
9: **end for**

Das Endmatching M wird extrem und perfekt sein, also besitzt dieses Matching minimales Gewicht (Satz 2.16).

Satz 2.18 *Der obige Algorithmus besitzt eine Aufwand von* $\mathcal{O}(n \cdot nnz(A) \cdot \log_2 n)$.

Die obere Grenze $\mathcal{O}(n \cdot nnz(A) \cdot \log_2 n)$ ist in Wirklichkeit zu pessimistisch. Dank "Implementierungstricks" läuft er viel schneller.

Wie auch früher besprochen, $M = \emptyset$ kann als extremes Anfangsmatching benutzt werden mit $u = v = (0, \ldots, 0)$. Folgendes elementares Lemma ist eine gute Grundlage für extreme Anfangsmatchings:

Lemma 2.6 *Sei* (V_r, V_c, E, c) *ein gewichteter bipartiter Graph. Sei* $|V_r| = |V_c| = n$ *und* $u, v \in \mathbb{R}^n$, *sodass* $(\forall (i,j) \in E)(u_i + v_j \leq c_{i,j})$. *Sei* $\widetilde{E} = \{(i,j) \in E \,|\, u_i + v_j = c_{i,j}\}$. *Dann ist jedes Matching in bipartiten Graphen* $(V_r, V_c, \widetilde{E})$ *extrem und zwar bezüglich* u *und* v.

[9] Es wird angenommen, dass unsere Anfangsmatrix A nicht strukturell singulär ist.

Folgende einfache Initialisierung für u und v ([19]) hat sich sehr gut bewährt:

$$(\forall i \in V_r) \quad u_i = \min_{j \in ROW(i)} c_{i,j}$$
$$(\forall j \in V_c) \quad v_j = \min_{i \in COL(j)} (c_{i,j} - u_i)$$

(Erinnerung: für $(i,j) \notin E$, $c_{i,j} = +\infty$) Da $u_i + v_j \leq u_i + (c_{i,j} - u_i) = c_{i,j}$, ist die Bediengung für Lemma 2.6 erfüllt. Benutzt man als (extremes) Anfangsmatching das, was sie liefert[10], erreicht man erhebliche Laufzeitverkürzungen.

Ich habe auch diesen Algorithmus implementiert. Die von Duff implementierten Algorithmen MC21, welcher (irgend) eine Transversale liefert, und MC64, welcher eine MPD Transversale liefert, sind allgemein bekannt ([25]). Im Folgenden habe ich meine MC21- und MC64-Implementierungen[11] mit den "echten" MC21 und MC64 verglichen. Die Testmaschine P4 HT 2.8GHz, Linux Suse 8.2. Kompiliert wurde mit den -O3 Option. Alle Angaben sind in Sekunden.

Matrix Name	Dim	MC21	MC64	my_MC21	my_MC64
lhr71	70304 × 70304	6.259750e-01	1.463666e+00	1.561431e-01	3.849548e+00
av41092	41092 × 41092	4.038946e+00	6.089425e+00	7.994570e-01	8.384678e+01
ibm_matrix_2	51448 × 51448	1.108901e-02	1.612507e+00	3.120103e-02	1.737409e-01
3dtube	45330 × 45330	4.648707e-02	2.321601e+00	1.656091e-01	1.248715e+00
lhr71c	70304 × 70304	2.298589e-01	3.961633e+00	1.456450e-01	1.546372e+01
mark3jac140	64089 × 64089	7.412803e-02	9.720439e-01	5.397792e-02	9.064685e+00
mark3jac140sc	64089 × 64089	7.277997e-02	1.143577e+00	5.422307e-02	6.160566e+00
matrix_new_3	125329 × 125329	2.577603e-02	2.109000e+00	5.490690e-02	3.045569e-01
matrix_9	103430 × 103430	2.683201e-02	1.993445e+00	6.827192e-02	4.614801e-01
onetone_1	36057 × 36057	3.320020e-01	8.993588e-02	1.238559e-01	4.268911e-01
onetone_2	36057 × 36057	2.739149e-01	6.778889e-02	2.719992e-02	3.938090e-01
pre2	659033 × 659033	2.021844e+01	3.265795e+00	7.620489e-01	1.088439e+02
rim	22560 × 22560	1.635030e-01	3.486480e-01	1.305379e-01	2.293032e+00
twotone	120750 × 120750	8.570633e+00	3.301459e-01	1.047620e-01	9.506320e+00
li	22695 × 22695	1.673607e-02	2.022259e-01	5.945091e-02	2.616989e-01
epb3	84617 × 84617	1.542612e-02	1.154199e-01	3.204611e-02	2.035220e-01
bbmat	38744 × 38744	2.648696e-02	3.931011e-01	9.006094e-02	3.754755e+00
boyd1	93279 × 93279	2.568507e-02	4.776336e+01	8.583003e-02	6.170363e+01
boyd2	466316 × 466316	1.983870e+03	6.956239e-01	2.117441e-01	2.944489e+02

2.5.2.3 Umwandlung zu $I - Matrix$

Definition 2.8 *Eine Matrix $B \in \mathbb{R}^{n \times n}$ heißt* **I – Matrix** *falls:*

- $(\forall i, j = i, \ldots, n)\ |B_{i,j}| \leq 1$
- $(\forall i = i, \ldots, n)\ B_{i,i} = 1$

Der Satz von Gershgorin besagt, dass die Eigenwerte einer Matrix $A \in \mathbb{C}^{n \times n}$

[10] Algorithmen für "normale" Matchings wurden schon besprochen.
[11] Ich nenne diese hier my_MC21 und my_MC64. Siehe 2.1 und 2.5.2 für Einzelheiten.

in der Vereinigung aller abgeschlossenen Kreisscheiben:

$$K_j = \{\xi \in \mathbb{C} \,|\, |\xi - a_{j,j}| \leq \sum_{k \neq j} |a_{j,k}|\}, \text{ für } j = 1, \ldots, n$$

liegen. Eine sparse $I - Matrix$ könnte[12] die Eigenwerte nah an 1 haben und das ist von Vorteil bei den Iterativen Methoden.

Satz 2.19 ([51]) *Jede strukturell nicht-singuläre Matrix $A \in \mathbb{R}^{n \times n}$ kann durch Skalierungen und eine Permutation zu einer $I - Matrix$ umgewandelt werden.*

Beweis: Seien $(a_i)_{i=1,\ldots,n}$ und $C = (c_{i,j})_{i,j=1,\ldots,n}$ wie bei 2.4 definiert. Sei M das MWT Matching bezüglich C, was eigentlich das MPD Matching bezüglich A ist und u,v die dazugehörigen Vektoren (gemäß 2.7). Konstruiere die Permutation ψ nach: $(i,j) \in M \Leftrightarrow j = \psi(i)$ und sei $Q = I(\psi, :)$, die zu ψ gehörige Permutationsmatrix. Definiere:

$$\widetilde{D} = \text{diag}(a_1, \ldots, a_n)$$
$$\widetilde{U} = \text{diag}(\exp(u_1), \ldots, \exp(u_n))$$
$$V = \text{diag}(\exp(v_1), \ldots, \exp(v_n))$$

Sei:

$$\widehat{B} = \widetilde{U}\widetilde{D}^{-1}AV$$

Dann gilt: $|\widehat{B}_{i,j}| = \exp(u_i - \log(a_i) + \log(|a_{i,j}|) + v_j) = \exp(u_i + v_j - c_{i,j}) \leq 1$ und $|\widehat{B}_{i,\psi(i)}| = 1$. Um $\widehat{B}_{i,\psi(i)} = 1$ zu haben, müssen wir das Vorzeichen von $a_{i,\psi(i)}$ berücksichtigen. Sei nun:

$$U = \widetilde{U} \cdot \text{diag}\big(\text{sign}\big(a_{1,\psi(1)}\big), \ldots, \text{sign}\big(a_{n,\psi(n)}\big)\big) =$$
$$= \text{diag}\big(e^{u_1}\,\text{sign}\big(a_{1,\psi(1)}\big), \ldots, e^{u_n}\,\text{sign}\big(a_{n,\psi(n)}\big)\big)$$
$$B' = U\widetilde{D}^{-1}AV$$

Jetzt gilt $|B'_{i,j}| \leq 1$ und $B'_{i,\psi(i)} = 1$. Also $B = B'(:, \psi) = B'Q^T$ ist eine $I - Matrix$. □

[12] obwohl es nicht garantiert ist

Kapitel 3

Fill-in reduzierende Permutationen

In diesem Kapitel wird nur die Struktur der Matrizen eine Rolle spielen. Also werden die Matrizen rein strukturell betrachtet. Eine tiefe Beschreibung der hier vorgestellten Methoden wird aus Platzgründen nicht möglich sein. Ich habe versucht, sie so kompakt wie möglich zu beschreiben.

Sei A eine $n \times n$ SPD Matrix und $A = LDL^T$ ihre Cholesky-Zerlegung. A priori angewandte symmetrische Permutationen spielen eine wichtige Rolle bei dieser Zerlegung. Dadurch wird es möglich, sowohl die Anzahl der arithmetischen Operationen als auch das Fill-in zu verkleinern. Je weniger arithmetische Operationen, desto weniger Rundungsfehler treten auf. Für den nicht SPD Fall, will man auch dass die LDU-Berechnung stabil bleibt, so muss man Pivots berücksichtigen. Dass wird fürs erste in diesem Kapitel weggelassen.

Es sollte erwähnt werden, dass die Aufgabe, eine Permutation zu finden, welche das Fill-in minimiert, ein NP-Vollständiges[1] Problem ist ([67]). So ist man gezwungen Heuristiken zu benutzen. Am Ende des Kapitels werden Tests aufgelistet, welche die Unerlässlichkeit einer Fill-in-verkleinernden Permutation bestätigen.

3.1 Die RCM-Permutation

Für eine beliebige Matrix B $(n \times n)$ (mit nullfreier Diagonale) definiere die **Enveloppe** als die Vektoren p_B (für die Zeilen) und q_B (für die Spalten):

$$(\forall i \in \{1, \ldots, n\})\ p_B(i) = \min\{j \mid (i,j) \in G(B)\}$$

[1] Der Autor in [67] zeigt, dass die Aufgabe *schwerer* ist als ein anderes, schon als NP-vollständig bekanntes Problem. Er zeigt nicht das die Aufgabe selbst zur Klasse NP gehört. Daher sollte statt NP-Vollständig besser NP-hart stehen (NPC-Definition nach [21], Seite 986).

$$(\forall j \in \{1,\ldots,n\})\; q_B(j) = \min\{i \mid (i,j) \in G(B)\}$$

Das **Profil** von B definiert man als:

$$\text{Profil}\,(B) = \{(i,j)\ :\ (p_B(i) \le j \le i) \vee (q_B(j) \le i \le j)\}$$

Satz 3.1 ([30]) *Sei $B = L_B D_B U_B$ die LDU Zerlegung. Dann gilt*

$$\text{Profil}\,(L_B + U_B) = \text{Profil}\,(B)$$

Sei nun A symmetrisch und $G(A)$ zusammenhängend. Wegen der Symmetrie gilt $q_A = p_A$. Satz 3.1 legt nahe, dass ein kleines Anfangsprofil von A ein kleines Profil von L impliziert und dadurch einen "kleinen" Fill-in.

Eine mögliche Herangehensweise, das Profil zu verkleinern, ist, die Einträge nah an die Diagonale zu permutieren. Das kann man erreichen, indem man die Matrix in eine blocktridiagonale Gestalt permutiert, wobei die Diagonalblöcke möglichst klein sind, und damit deren Anzahl entsprechend groß.

Diese Herangehensweise führt zu dem Begriff der **Levelsets** (oder Niveaumengen):

Definition 3.1 *Sei $G = (V,E)$ ein (ungerichteter) zusammenhängender Graph und $S_0 \subset V$ eine nichtleere Knotenmenge. Die zu S_0 gehörigen **Levelsets** $S_0, S_1, S_2,$ $\ldots$ sind wie folgt definiert:*

$$\begin{aligned}
S_1 &= \{v\ :\ (\exists w \in S_0)(\{v,w\} \in E)\}\backslash S_0\\
S_2 &= \{v\ :\ (\exists w \in S_1)(\{v,w\} \in E)\}\backslash (S_0 \cup S_1)\\
&\vdots\\
S_k &= \{v\ :\ (\exists w \in S_{k-1})(\{v,w\} \in E)\}\backslash (S_0 \cup S_1 \cup \ldots \cup S_{k-1})\\
&\vdots
\end{aligned}$$

Wie man leicht einsieht, gilt:

Lemma 3.1 *Für $k = 0,1,2,\ldots,$ ist S_k die Menge aller Knoten, deren Abstand von S_0 genau k ist.*

Offensichtlich gibt es einen Index k_0, sodass $S_k = \emptyset$ für $k > k_0$. Sei $\widetilde{k}$ der kleinste k_0.

Lemma 3.2

a) V ist in $\widetilde{k}+1$ nichtleere Teilmengen partitioniert.

b) $\widetilde{k} \le \text{Diam}\,(G) < n$

c) Jede Kante aus G verbindet entweder zwei Knoten aus demselben Levelset oder aus zwei aufeinanderfolgenden Levelsets.

Sei jetzt $G = G(A)$. Jede Permutation π, welche die Knoten in $S_0, S_1, \ldots, S_{\widetilde{k}}$ nacheinander nummeriert, wobei es egal ist, wie die Knoten innerhalb jedes Levelset nummeriert werden, bewirkt eine Blocktridiagonalgestalt der permutierten Matrix

$$P_\pi A P_\pi^T = \begin{pmatrix} B_0 & C_0 & & \\ C_0 & B_1 & C_1 & \\ & C_1 & \ddots & C_{\widetilde{k}-1} \\ & & C_{\widetilde{k}-1} & B_{\widetilde{k}} \end{pmatrix}$$

mit $\widetilde{k}+1$ Diagonalblöcken.

In der Praxis enthält S_0 einen einzelnen Knoten. Ideal wäre es wenn $\widetilde{k} = \text{Diam}\,(G(A))$. Es ist nicht bekannt wie man zwei diametrale Knoten in einem Graphen in $\mathcal{O}(n + \text{nnz}(A))$ findet. Man kann aber versuchen, *pseudoperiphere* Knoten zu finden.

Der Cuthill-Mckee Algorithmus (kurz CM) baut die Levelsets aus einer gegebene Anfangsmenge S_0 mit Aufwand $\mathcal{O}(n + \text{nnz}(A))$ auf ([48]):

Algorithmus 3.1 CM - Anordnung

Beschreibung: Input: $G = (V, E)$ zusamenhängend, $S_0 = \{v\}$; Output: π, *level*

```
1: level(1 : n) = −1, i = 1, π(1) = v, level(v) = 0
2: Sei Q eine leere Queue, enque(S, v)
3: while Q ≠ ∅ do
4:    w = dequeue(Q)
5:    for {w, u} ∈ E do
6:       if level(u) == −1 then {u noch nicht entdeckt}
7:          i + +
8:          π(i) = u, level(u) = level(w) + 1
9:          enqueue(S, u)
10:      end if
11:   end for
12: end while
```

Satz 3.2 ([48]) *Die Mengen $S_k = \{w \in V : level(w) = k\}$, $k = 0, 1, \ldots, max(level)$, sind die Levelmengen zur Anfangsmenge $S_0 = \{v\}$. Sei $B = P_\pi A P_\pi^T$. Dann besitzt B blocktridiagonale Gestalt. Außerdem gilt für die Enveloppe $p_B(i) \leq p_B(i+1)$, $\forall i = 1, 2, \ldots, n-1$.*

Somit besitzt die Enveloppe der permutierten Matrix eine “Treppengestalt“. Wie Abbildung 3.1 zeigt, kann man mit einem “netten Trick“ die Enveloppe

nochmals verkleinern. Wendet man bei der CM-Permutation noch eine Spiegelung bezüglich der Antidiagonalen an, so erhält man die sog. Reverse CM-, kurz RCM-Anordnung. Im Abbildung 3.1 ist diese gespiegelte Matrix als $C = P_\psi A P_\psi^T$ bezeichtet, wobei die Permutation der RCM-Anordnung ψ gegeben ist durch:

$$(\forall i = 1, 2, \ldots, n)\ \psi(i) = \pi(n + 1 - i)$$

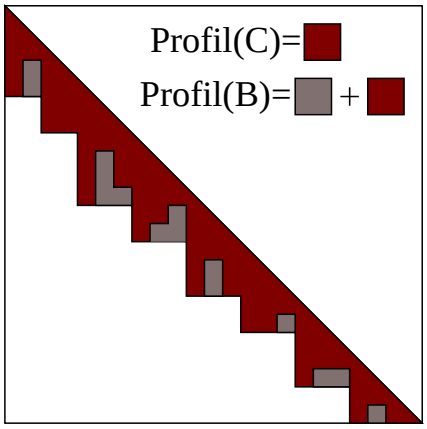


Abbildung 3.1:

3.2 Die MinDeg-Permutation

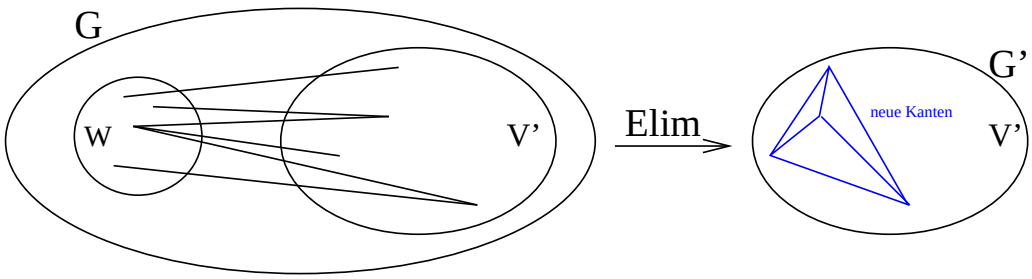
Sei $G = (V, E)$ ein ungerichteter Graph. Sei $W \subset V$. Folgender Operator **Elim** modelliert die Elimination aller Knoten $w \in W$ aus G:

$$G' = \mathrm{Elim}_G(W) = (V', E') \quad \text{mit}$$

$$V' = V \backslash W \quad \text{und}$$

$$E' = (E \backslash \{\{u, v\} \in E\ :\ (u \in W) \vee (v \in W)\}) \bigcup \{\{u, v\}\ :\ u, v \in \mathrm{Adj}_G(W)\}$$

Elim entfernt alle Kanten, die aus W entspringen, und fügt die Clique mit den Knoten $\mathrm{Adj}_G(W) \subseteq V'$ hinzu[2]. Ist $W = \{w\}$ einelementig, statt $\mathrm{Elim}_G(\{w\})$ werden wir $\mathrm{Elim}_G(w)$ schreiben.



[2]Es ist möglich, dass einiger dieser Kanten bereits in G vorhanden waren.

Ein allgemeiner Eliminationsprozess würde so aussehen:

Algorithmus 3.2 Allgemeine Eliminationsprozess

Beschreibung: Input: $G = (V, E)$, $n = |V|$; Output: π

1: **for** $k = 1 : n$ **do**
2: Wähle (Pivot) $w \in V$ nach irgendeinem Kriterium
3: $\pi(k) = w$
4: $G = \mathrm{Elim}_G(w)$ {Datiere G auf}
5: **end for**

Die Anordnung $\pi = (\pi_1, \pi_2, \ldots, \pi_n)$ heißt auch *Eliminationsreihenfolge*. Entsprechend des Pivotkriteriums (Zeile 2), erhält man verschiedene Eliminationsreihenfolgen. Eine davon ist die Minimum-Degree Anordnung ([48]):

Definition 3.2 *Wählt man in Algorithmus 3.2 in Zeile 2 immer den Pivot w mit dem kleinsten Grad $\mathrm{Deg}_G(w) = \min\{\mathrm{Deg}_G(v) : v \in V\}$, so heißt die resultierende Permutation Minimum-Degree Anordnung, kurz MD. Eine MD ist normalerweise nicht eindeutig.*

MD versucht durch eine greedy Strategie das Fill-in zu minimieren. Ein allgemeiner MD-Algorithmus kann so formuliert werden:

Algorithmus 3.3 Allgemeine MD

Beschreibung: Input: $G = G_1 = (T_1, E_1)$, $n = |T_1|$; Output: π

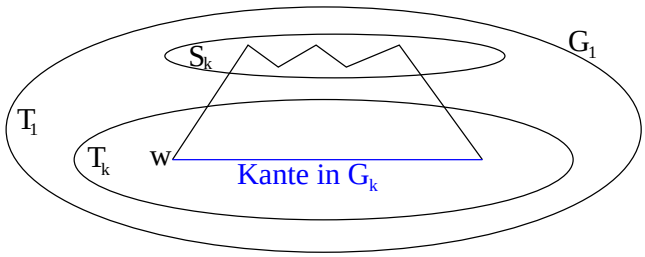
1: **for** $k = 1 : n$ **do**
2: Wähle $\pi_k \in T_k$ mit dem kleinsten Grad: $\mathrm{Deg}_{G_k}(\pi_k) = \min\{\mathrm{Deg}_{G_k}(w) : w \in T_k\}$
3: $G_{k+1} = (T_{k+1}, E_{k+1}) = \mathrm{Elim}_{G_k}(\pi_k)$
4: **end for**

Die einzelnen Graphen $G_k = (T_k, E_k)$ heißen **Eliminationsgraphen**. Eine naive Implementierung bräuchte großen Speicher, denn die Kantenmengen E_k wachsen an (bis zu einem Zeitpunkt, wo G_k fast eine Clique wird). Es gibt aber eine sehr feine Modellierung des Eliminationsprozesses durch den *Quotienten-Eliminationsgraphen*, welche keinerlei zusätzlichen Speicher braucht. Folgender Satz[3] bereitet den Weg zum Quotienten-Eliminationsgraphen:

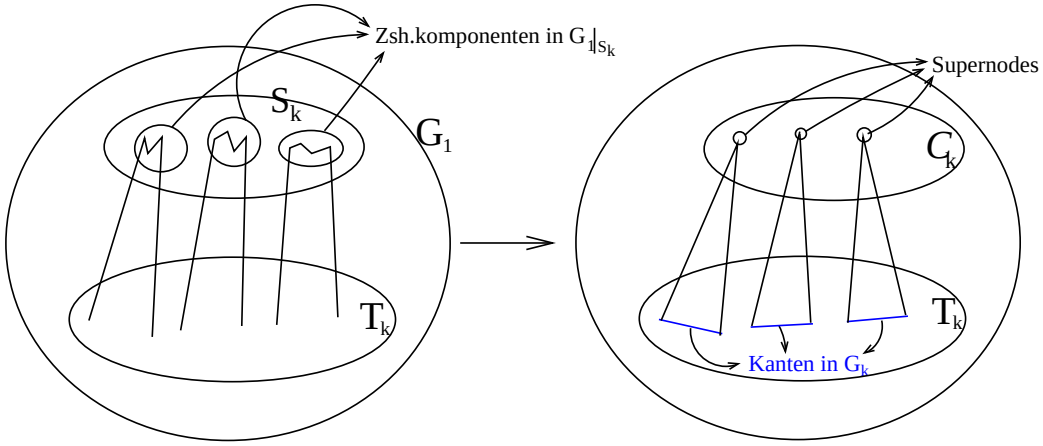
Satz 3.3 ([48]) *Definiere $(\forall k = 1, 2, \ldots, n)\ S_k = \{\pi_1, \pi_2, \ldots, \pi_{k-1}\} = \{1, 2, \ldots, n\} \backslash T_k$. Dann gilt:*

$$(\forall w \in T_k)\ \mathrm{Adj}_{G_k}(w) = \mathrm{Reach}_{G_1}(w, S_k)$$

[3] Für ein Beweis siehe Korollar 4.1.



Also würde man die Zusammenhangskomponenten von $G_1\Big|_{S_k}$ als einzelne Knoten komprimieren (sog. Superknoten), so wären die Nachbarknoten jedes beliebigen Knoten $w \in T_k$ in G_k entweder Nachbarknoten aus G_1 oder über einen Pfad der Länge 2 über S_k zu erreichen.



Definition 3.3 *Sei $G = (V, E)$ ein Graph, π eine Eliminationsreihenfolge, $S_k = \{\pi_1, \pi_2, \ldots, \pi_{k-1}\}$ und $T_k = \{1, 2, \ldots, n\} \backslash S_k$. Sei*

$$\mathcal{C}(S_k) = \{\mathcal{C} : \mathcal{C} \subseteq V \text{ ist Zsh-Komponente von } G\Big|_{S_k}\}$$

und

$$\mathcal{V}_k = \mathcal{C}(S_k) \bigcup \{\{w\} : w \in T_k\}.$$

*Der k-te **Quotienten-Eliminationsgraph** $\mathcal{G}_k$ ist definiert als der Quotientengraph $\mathcal{G}_k = G/\mathcal{V}_k$.*

Unter Bezugnahme auf Quotienten-Eliminationsgraphen kann der Satz 3.3 wie folgt formuliert werden:

Satz 3.4 ([48])

$$(\forall w \in T_k)\ \ \mathrm{Adj}_{G_k}(w) = \mathrm{Reach}_{G_1}(w, S_k) = \mathrm{Reach}_{\mathcal{G}_k}([w], \mathcal{C}(S_k))$$

Die letzte Gleichung ist eigentlich Unsinn, denn die Knoten von G_1 sind aus V, die von $\mathcal{G}_k$ aus der Potenzmenge von V. Es ist aber verständlich, was man mit der Gleichheit ausdrücken will.

Da $\mathcal{G}_k\Big|_{\mathcal{C}(S_k)}$ nur isolierte Knoten enthält (keinerlei Kanten), ist die Menge

$$\text{Reach}_{\mathcal{G}_k}([w], \mathcal{C}(S_k))$$

und dadurch auch die Menge $\text{Adj}_{G_k}(w)$, ziemlich einfach zu bestimmen. Der MD-Algorithmus sieht unter Verwendung von Quotienten-Eliminationsgraphen so aus:

Algorithmus 3.4 MD Algorithmus, mit Quotienteneliminationsgraphen

Beschreibung: Input: $G = (V, E)$, $n = |V|$; Output: π

1: $deg = zeros(1, n)$
2: **for** k=1:n **do**
3: $deg(k) = \text{Deg}_G(k)$
4: **end for**
5: $S_1 = \emptyset$, $T_1 = V$, $\mathcal{G}_1 = G$, $\mathcal{C}(S_1) = \emptyset$
6: **for** $k = 1 : n$ **do**
7: Wähle $\pi_k \in T_k$ mit dem kleinsten Grad: $deg(\pi_k) = \min\{deg(w) : w \in T_k\}$
8: Bestimme $R = \text{Reach}_{\mathcal{G}_k}(\pi_k, \mathcal{C}(S_k))$
9: {Konstruiere $\mathcal{G}_{k+1}$ aus $\mathcal{G}_k$}
10: Bilde eine neue Supernode aus π_k und $(\text{Adj}_{\mathcal{G}_k}(\pi_k) \cap \mathcal{C}(S_k))$ und lösche eventuell doppelt vorhandene Kanten die dadurch entstehen
11: $S_{k+1} = S_k \cup \{\pi_k\}$, $T_{k+1} = T_k \backslash \{\pi_k\}$
12: $\{\mathcal{C}(S_{k+1}) = (\{\pi_k\} \cup \mathcal{C}(S_k)) \setminus (\text{Adj}_{\mathcal{G}_k}(\pi_k) \cap \mathcal{C}(S_k))\}$
13: {In einige Implementierungen, z.B AMD, werden alle Kanten mit Endpunkte in R aus $\mathcal{G}_{k+1}$ entfernt}
14: {$\mathcal{G}_{k+1}$ ist jetzt konstruiert}
15: **for** $w \in R$ **do**
16: Datiere $deg(w)$ auf z.B. mittels Satz 3.4
17: **end for**
18: **end for**

Folgender Satz zeigt, dass zur Speicherung der $\mathcal{G}_k$'s nicht mehr Speicherplatz benötigt wird als für G:

Satz 3.5 ([48]) *Sei $\mathcal{G}_k = (\mathcal{V}_k, \mathcal{F}_k)$. Für $k = 1, 2, \ldots, n-1$ gilt:*

$$|\mathcal{F}_{k+1}| \leq |\mathcal{F}_k| \leq |E|$$

Im Abblidung 3.2 sind die einzelnen Schritte für einen Graphen mit 10 Knoten dargestellt. Es wird sowohl mit Eliminationsgraphen als auch mit Quotienteneliminationsgraphen vorgegangen. Gelbe Knoten stellen Supernodes dar.

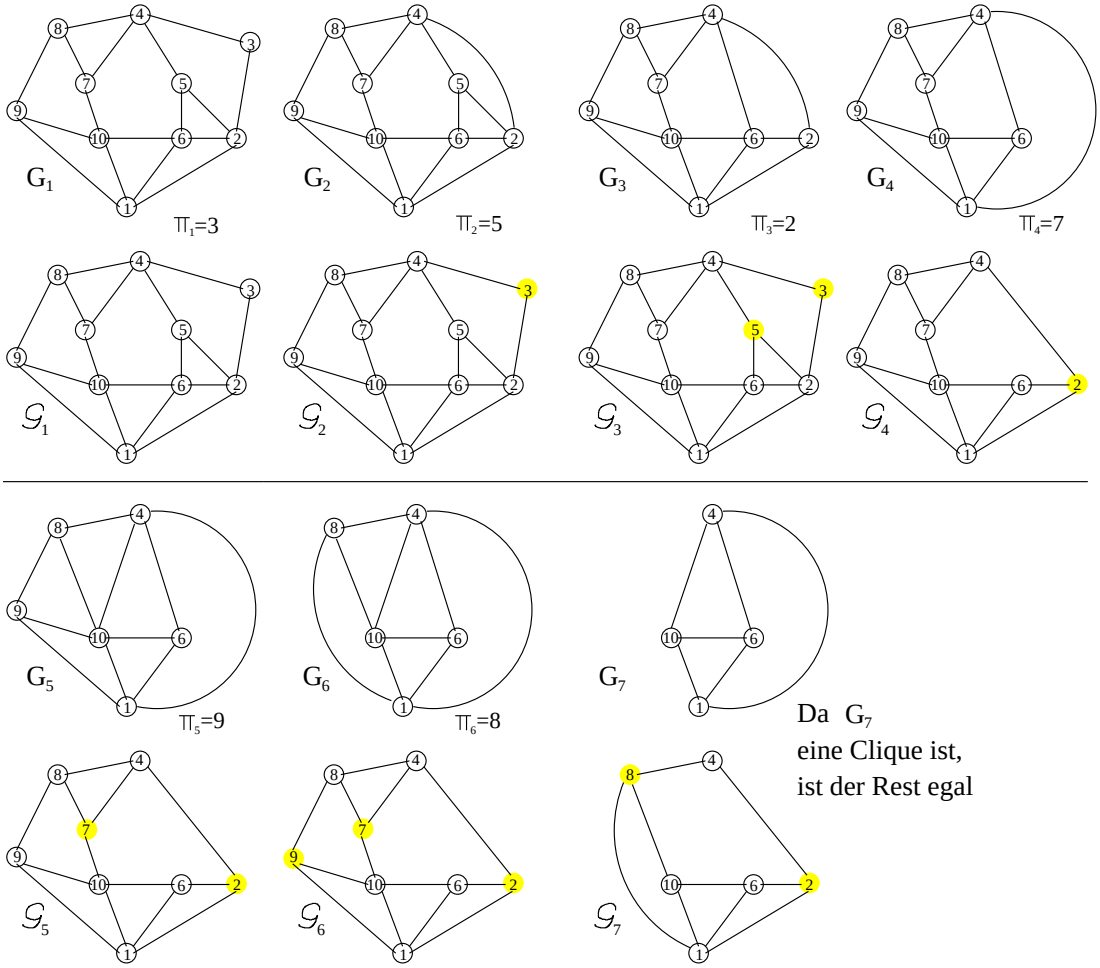


Abbildung 3.2: Beispiel für eine Graph mit 10 Knoten

Mit diesem Algorithmus ist das Speicherproblem zwar gelöst, aber dafür haben wir ein neues geschaffen, nämlich die Aktualisierung der Grade von $w \in R$ (Zeilen 15,16,17). Es folgen einige bewährte Techniken um diese Phase zu beschleunigen ([36] und [44]).

3.2.1 Beschleunigungstechniken für MD

3.2.1.1 Masseneliminination

Definition 3.4 *Zwei Knoten y, z eines Graphen $G = (V, E)$ heißen* ***nicht unterscheidbar*** *in G, falls*

$$\mathrm{Adj}_G(y) \cup \{y\} = \mathrm{Adj}_G(z) \cup \{z\}$$

Wird die obere Bedingung nicht erfüllt, so sind die Knoten **unterscheidbar**.

Nicht unterscheidbare Knoten haben den gleichen Grad (triviale hash-Funktion). Folgendes Lemma ist elementar zu beweisen:

Lemma 3.3 *Seien y und z nicht unterscheidbar in G und $x \neq y, z$ ein Knoten aus G. Dann sind y und z auch nicht unterscheidbar in* $\mathrm{Elim}_G(x)$.

Korollar 3.1 *Seien y und z nicht unterscheidbar in G. Besitzt y minimale Grad in G, so besitzt z minimalen Grad in $G_y = \mathrm{Elim}_G(y)$, nämlich*

$$\mathrm{Deg}_{G_y}(z) = \mathrm{Deg}_G(y) - 1$$

Werden also zwei oder mehrere Knoten während des Eliminationsprozesses nicht unterscheidbar[4], so kann man diese zu einem einzigen (Super) Knoten komprimieren. Alle nachfolgenden Operationen brauchen nur auf diesen (Super) Knoten angewendet zu werden, und, wenn die "Eliminationszeit gekommen ist", eliminiert man alle Knoten des Superknotens auf einmal. Daher heißt diese Technik Masseneliminination.

Folgende Abbildung zeigt einen 5×5 Gittergraph G mit **9 Point Connectivity**, desen Knoten nach einer MD-Anordnung nummeriert worden sind. Nebenan ist die Lage nach 14 Schritten dargestellt, genauer gesagt $\mathcal{G}_{15}$. Die Menge $\mathcal{C}(S_{15})$ besteht aus vier in gelb gefärbte Supernodes.

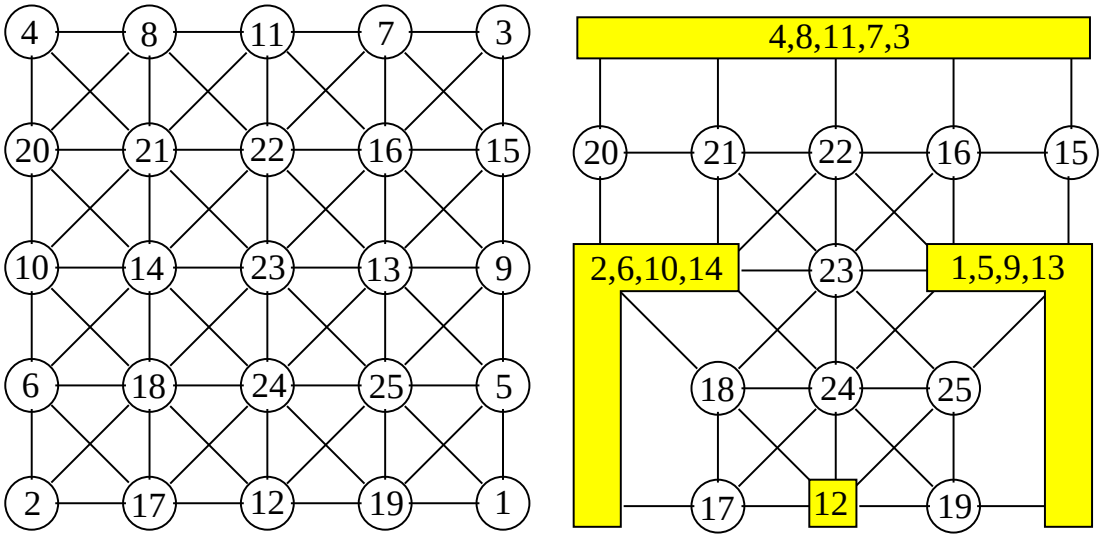


Abbildung 3.3: MD-Anordnung und das Zwischenstadium nach 14 Schritten

G_{15} ist nicht explizit angegeben, aber durch den Satz 3.4 können wir alle Nachbarschaftsinformationen herausfinden, z.B.

$$\mathrm{Adj}_{G_{15}}(16) = \mathrm{Reach}_{\mathcal{G}_{15}}(16, \mathcal{C}(S_{15})) = \{15, 19, 20, 21, 22, 23, 24, 25\}$$

[4] diese bilden zwangsweise eine Clique

$$\mathrm{Adj}_{G_{15}}(15) = \mathrm{Reach}_{\mathcal{G}_{15}}(15, \mathcal{C}(S_{15})) = \{16, 19, 20, 21, 22, 23, 24, 25\}$$

Also sind die Knoten $\{15, 16\}$ nicht unterscheidbar in G_{15}. Das Gleiche erhält man auch für die Mengen $\{17, 18\}$, $\{19, 25\}$, $\{20, 21\}$ und $\{22, 23, 24\}$. Diese 5 Mengen kann man als 5 Supernodes betrachten. (Offensichtlich wurde bei unseren MD-Anordnung keine systematische Masseneliminationen gemacht, denn 19 und 25 sind keine aufeinanderfolgenden Zahlen.)

3.2.1.2 Unvollständiger Degree-Update

Definition 3.5 *Seien y, z zwei Knoten eines Graphen $G = (V, E)$. Man sagt, dass der Knoten z von Knoten y in G* ***übertroffen*** *wird, falls*

$$\mathrm{Adj}_G(y) \cup \{y\} \subseteq \mathrm{Adj}_G(z) \cup \{z\}$$

Wird z von y übertroffen, so ist der Grad von y nicht größer als der Grad von z. Folgendes Lemma ist ebenfalls elementar zu beweisen:

Lemma 3.4 *Sei z von y in G übertroffen und $x \neq y, z$ ein Knoten aus G. Dann wird z von y auch in $\mathrm{Elim}_G(x)$ übertroffen.*

Korollar 3.2 *Sei z von y in G übertroffen. Dann kann der Knoten y während des MD-Algorithmus, vor dem Knoten z eleminiert werden.*

Wird also ein Knoten z während des MD-Eliminationsprozesses von einem anderen Knoten y übertroffen, so kann man auf Degree-Updates für z *verzichten*, solange y noch nicht eliminiert worden ist.

Am Beispiel der Abb. 3.3 sieht man, dass alle Knoten aus $\{22, 23, 24\}$ (in G_{15}) von jeden Knoten aus $\{15, 16, 17, 18, 19, 20, 21, 25\}$ übertroffen werden. (Offensichtlich wird die Tatsache, dass 25 die Knoten $\{22, 23, 24\}$ übertrifft, in unseren MD-Nummerierung nicht genutzt, denn $22, 23, 24 < 25$).

3.2.1.3 Multiple elimination

Sei Y eine unabhängige Knotenmenge aus G, sodass jedes Element $y \in Y$ minimalen Grad, bezüglich G, besitzt. Die Idee ist, alle Knoten aus Y auf einmal zu eliminieren, und erst danach nötige Degree-Updates in $\mathrm{Adj}_G(Y)$ durchzuführen. Dadurch können wir uns mehrfache Degree-Updates einsparen.

Diese Technik, multiple elimination genannt, kann bei gleichzeitiger Anwendung von Masseneliminationen mit echten Graden (nicht mit externen Graden, siehe Abschnitt 3.2.1.4), dazu führen, dass das Ergebnis kein echtes MD ist ([44]). Die Permutationsgüte ändert sich aber nur minimal oder bleibt ganz erhalten. Die Zeit, die dadurch erspart wird, rechtfertigt diese Modifikation.

Als Beispiel nehmen wir wieder den oben erwähnten 5×5 Gittergraph G. Nebenan ist die Lage nach 8 Schritten dargestellt, genauer gesagt $\mathcal{G}_9$. Die Menge $\mathcal{C}(S_9)$ besteht aus 4 Supernodes, in gelb gefärbt.

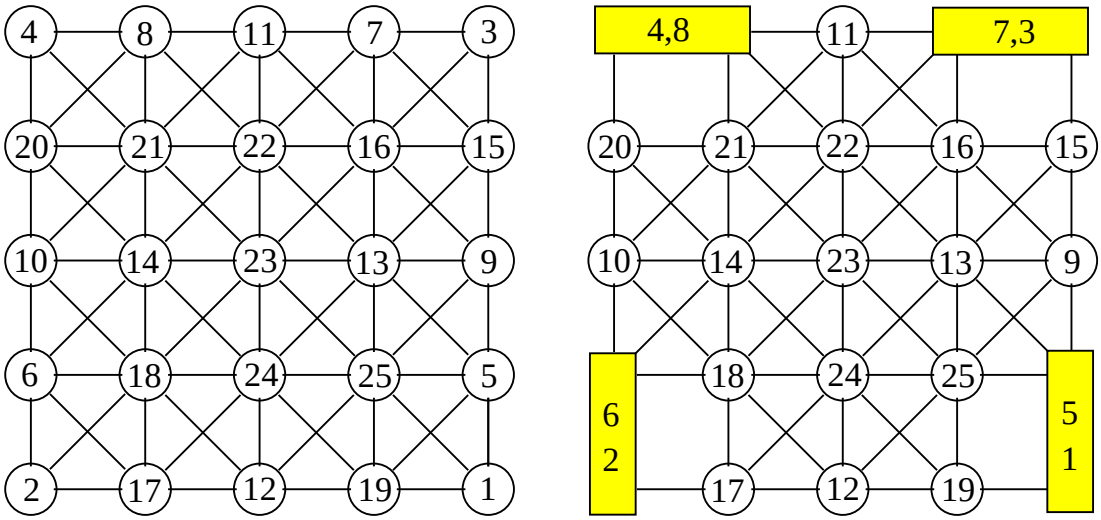


Abbildung 3.4: MD-Anordnung und das Zwischenstadium nach 8 Schritten

Nach 8 Schritten ist der minimale Grad 5. Alle Knoten, die zur unabhängigen Menge $\{9, 10, 11, 12\}$ gehören, besitzen Grad 5. Wendet man die multiple elimination Technik an, so erspart man sich doppelte Degree-Updates bei der Knotenmenge $\{15, 16, 17, 18, 19, 20, 21, 25\}$.

3.2.1.4 Externer Grad statt normalem Grad

Definition 3.6 *Sei $G = (V, E)$ ein Graph und Y eine Menge*[5] *nicht unterscheidbarer Knoten in G. Für $y \in Y$, definiert man den* ***externen Grad*** *als die Anzahl der adjazenten Knoten von y, welche von y unterscheidbar sind. Der externe Grad ist also einfach $|\operatorname{Adj}_G(Y)|$.*

Bei der MD arbeitet man standardmäßig mit normalen Graden. Möchte man aber Supernodes für die Masseneliminination nutzen (siehe 3.2.1.1), macht es durchaus Sinn, externen statt normalen Grad zu benutzen. Bei einelementigen Mengen stimmen externer und normaler Grad überein. Also kann man während des ganzen MD-Algorithmus nur externen Grad benutzen. Diese Technik kann dazu führen, dass das Ergebnis kein echtes MD ist. Experimente haben sogar gezeigt, dass die Benutzung des externen Grades das Fill-in etwas stärker reduziert als die Benutzung des normalen Grades.

[5] Y könnte auch ein einziges Element haben

Im Abb. 3.3, sind nach 14 Schritten die Knoten 15 und 16 nicht unterscheidbar:

$$\mathrm{Adj}_{G_{15}}(15) \cup \{15\} = \mathrm{Adj}_{G_{15}}(16) \cup \{16\} = \{15, 16,\ 19, 20, 21, 22, 23, 24, 25\}$$

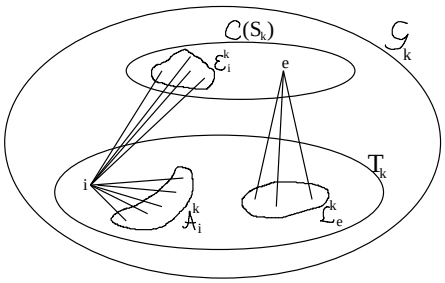
Ihr normaler Grad ist 8, ihr externer Grad ist 7.

3.3 Approximate Minimum Degree - Permutation (AMD)

Wie der Name schon sagt, ist auch AMD ([6]) kein "richtiger" MinDeg-Algorithmus. Das Ergebnis ist vergleichbar mit echten MD's, manchmal sogar besser. Vor allem ist er sehr schnell. Da ich ihn in meinem Code eingebunden habe, werde ich ihn etwas detaillierter beschreiben.

AMD benutzt die MD-Variante mit Quotienteneliminationsgraphen. Für die AMD ist eine tiefere Beschreibung des MD (mit Quotienteneliminationsgraphen) nötig. Als Basis dient die Beschreibung aus 3.2.

Eliminierte Knoten (die Supernodes aus $\mathcal{C}(S_k)$) heißen hier **Elemente** und die nicht eliminierten (die Knoten aus T_k) heißen **Variablen**. Sei $G = G_1 = (V, E)$ der Anfangsgraph.



Sei $\mathcal{G}_k = (\mathcal{V}_k, \mathcal{F}_k)$ der k-te Quotienteneliminationsgraph. Für $i \in T_k$, sei:
$\mathcal{A}_i^k = \{j \in T_k : \{i, j\} \in \mathcal{F}_k\}$,
$\mathcal{E}_i^k = \{e \in \mathcal{C}(S_k) : \{i, e\} \in \mathcal{F}_k\}$
mit $\mathrm{Adj}_{\mathcal{G}_k}(i) = \mathcal{A}_i^k \cup \mathcal{E}_i^k$.
Es gilt $\mathcal{A}_i^1 = \{j : \{i, j\} \in E\}$ und $\mathcal{A}_i^k \supseteq \mathcal{A}_i^{k+1}$.
Für $e \in \mathcal{C}(S_k)$, sei:
$\mathcal{L}_e^k = \mathrm{Adj}_{\mathcal{G}_k}(e) = \{i : \{i, e\} \in \mathcal{F}_k\} \subseteq T_k$

Mit dieser Notationen besagt der Satz 3.5 (der Speicherplatz reicht):

$$\sum_{i \in T_k} |\mathcal{A}_i^k| + \sum_{i \in T_k} |\mathcal{E}_i^k| + \sum_{e \in \mathcal{C}(S_k)} |\mathcal{L}_e^k| \leq \sum_{i \in T_1} |\mathcal{A}_i^1|$$

Nach dem Satz 3.4 folgt

$$(\forall i \in T_k)\ \ \mathrm{Adj}_{G_k}(i) = \left(\mathcal{A}_i^k \cup \bigcup_{e \in \mathcal{E}_i^k} \mathcal{L}_e^k \right) \setminus \{i\}$$

AMD benutzt Masseneliminination. Nichtunterscheidbarkeit auf $\mathcal{G}_k$ impliziert Nichtunterscheidbarkeit auf G_k, das Gegenteil stimmt aber nicht. Da das Testen

auf Nichtunterscheidbarkeit auf $\mathcal{G}_k$ wesentlich einfacher ist als das Testen auf Nichtunterscheidbarkeit auf G_k, wird bei AMD auf Nichtunterscheidbarkeit auf $\mathcal{G}_k$ getestet (wenn überhaupt). Um Nichtunterscheidbarkeiten zu entdecken, wird folgende hash Funktion benutzt:

$$\text{Hash}(i) = \left[\left(\sum_{j\in\mathcal{A}_i^k} j + \sum_{e\in\mathcal{E}_i^k} e\right) \mod (n-1)\right] + 1$$

Haben zwei Supervariablen $\mathbf{i}$ und $\mathbf{j}$ gleiche Hashwerte $\text{Hash}(i) = \text{Hash}(j)$, so werden sie auf Nichtunterscheidbarkeit geprüft. Sie müssen noch nicht einmal mit einer Kante in $\mathcal{G}_k$ verbunden sein, solange sie einen gemeinsamen Nachbarn in $\mathcal{C}(S_k)$ haben (diese Kante ist überflüssig). Sollten sie wirklich nicht unterscheidbar sein, dann werden sie zu einer einzelnen Supervariable verschmolzen.
Durch Massenelimination werden die Variablen zu Supervariablen, d.h. die Knoten aus T_k sind in Supervariablen gruppiert, wobei auch *einelementige* Supervariablen erlaubt sind. Supervariablen werden in fetter Schrift gedruckt. Da jede Supervariable einen Repräsentanten braucht, bezeichnen wir eine Supervariable $\mathbf{i}$ nach ihren Repräsentant $i \in \mathbf{i}$. Für einelementige Supervariablen gilt $\mathbf{i} = \{i\}$.
AMD benutzt externe Grade; genauer gesagt, Näherungswerte für diese Grade. Im Schritt k ist der genaue externe Grad einer Supervariable mit Repräsentant i gegeben durch:

$$d_i^k = |\operatorname{Adj}_{G_k}(i)\backslash\mathbf{i}| = \left|\left(\mathcal{A}_i^k \cup \bigcup_{e\in\mathcal{E}_i^k} \mathcal{L}_e^k\right)\backslash\mathbf{i}\right| \tag{3.1}$$

Wie der Näherungswert $\overline{d}_i^k$ für d_i^k berechnet wird, wird später erläutert.
Zeile 13 des MD-Algorithmus auf Seite 49 ist bemerkenswert. Kanten die Knoten verbinden, welche über $\mathcal{C}(S_k)$ verbindbar sind, werden entfernt. Auf unseren Notationen kann es wie folgt aufgefasst werden:

$$(\forall k = 1,\ldots,n)(\forall i \in T_k)\ \left(\mathcal{A}_i^k \cap \left(\bigcup_{e\in\mathcal{E}_i^k} \mathcal{L}_e^k\right) = \emptyset\right)$$

Dadurch geht uns überhaupt keine Nachbarschaftsinformation (bezüglich G_k) verloren. Im Gegenteil wird dadurch die Berechnung der Näherungswerte $\overline{d}_i^k$ einfacher und genauer.
Multiple elimination wird auch angewandt, aber die Autoren sind auf dieses Thema nicht näher eingegangen. Auch im folgenden Algorithmus wird es nicht erwähnt. Warum "unvollständige Degree Updates" (3.2.1.2) nicht ausgenutzt werden kann, wird später kurz erklärt.
Im folgenden Algorithmus, S repräsentiert S_k, $\overline{S}$ repräsentiert $\mathcal{C}(S_k)$, T repräsentiert T_k und $\overline{T}$ ist die Menge der Repräsentanten aller Supervariablen. Das Gerüst für den (A)MD-Algorithmus[6] sieht so aus:

[6] es wird immer noch mit genauen (externen) Graden gearbeitet

Algorithmus 3.5 Gerüst für (A)MD

Beschreibung: Input: $G = (V, E)$, $n = |V|$; Output: π

1: $S = \overline{S} = \emptyset$, $T = \{1, 2, \ldots, n\}$
2: **for** i=1:n **do**
3: $\mathcal{A}_i = \{j : \{i, j\} \in E\}$
4: $\mathcal{E}_i = \emptyset$
5: $d_i = |\mathcal{A}_i|$ {es wird mit externen Grade gearbeitet}
6: $\mathbf{i} = \{i\}$ {macht Supervariablen aus Variablen}
7: **end for**
8: $\overline{T} = \{1, 2, \ldots, n\}$ {enthält die Repräsentanten aller Supervariablen}
9: $k = 1$
10: **while** $k \leq n$ **do**
11: *Masseneliminination:*
12: Wähle $p \in \overline{T}$ mit den kleinsten (externen) Grad d_p
13: $\pi(k : k + |\mathbf{p}| - 1) = \mathbf{p}$
14: $\mathcal{L}_p = \left(\mathcal{A}_p \cup \bigcup_{e \in \mathcal{E}_p} \mathcal{L}_e\right) \backslash \mathbf{p}$
15: **for** $i \in \mathcal{L}_p \cap \overline{T}$ **do** {d.h. alle $i \in \mathcal{L}_p$ die Repräsentanten sind}
16: $\mathcal{A}_i = (\mathcal{A}_i \backslash \mathcal{L}_p) \backslash \mathbf{p}$ { überflüssige Kanten werden entfernt; Vgl. mit Zeile 13 des MD-Alg. auf Seite 49}
17: $\mathcal{E}_i = (\mathcal{E}_i \backslash \mathcal{E}_p) \cup \{p\}$ {da p und $\mathcal{E}_p$ zu neuen (super)Elementen p verschmolzen werden}
18: $d_i = |\mathcal{A}_i| + \left|\left(\bigcup_{e \in \mathcal{E}_i} \mathcal{L}_e\right) \backslash \mathbf{i}\right|$ {externe Grad wird berechnet}
19: **end for**
20: **for** $e \in \mathcal{E}_p$ **do**
21: $\mathcal{L}_e = \emptyset$ {Da e durch p ersetzt wird}
22: **end for**
23: *mittels eine hash Funktion, suche in $\mathcal{L}_p$ nach Nichtunterscheidbarkeiten:*
24: **for** $i \in \mathcal{L}_p \cap \overline{T}$ **do**
25: $\text{Hash}(i) = \left[\left(\sum_{j \in \mathcal{A}_i} j + \sum_{e \in \mathcal{E}_i} e\right) \mod (n-1)\right] + 1$
26: **end for**
27: **for** $i, j \in \mathcal{L}_p \cap \overline{T}$ mit $\text{Hash}(i) = \text{Hash}(j)$ **do** {i und j sind nicht verbunden}
28: **if** $\mathcal{A}_i = \mathcal{A}_j$ und $\mathcal{E}_i = \mathcal{E}_j$ **then** {(Super)Variablen i und j nicht unterscheidbar in $\mathcal{G}_k$}
29: *verschmelze sie in eine Supervariable mit Repräsentant i:*
30: $\mathbf{i} = \mathbf{i} \cup \mathbf{j}$
31: $d_i = d_i - |\mathbf{j}|$ {datiere externen Grad auf}
32: $\overline{T} = \overline{T} \backslash \{j\}$ {j ist keine Repräsentante mehr}
33: $\mathcal{A}_j = \mathcal{E}_j = \emptyset$ {da alle Variablen aus $\mathbf{j}$ von i repräsentiert werden}
34: **end if**
35: **end for**
36: *mache aus Variable p eine Element p:*
37: $\overline{S} = (\overline{S} \cup \{p\}) \backslash \mathcal{E}_p$, $S = S \cup \mathbf{p}$
38: $\overline{T} = \overline{T} \backslash \{p\}$, $T = T \backslash \mathbf{p}$
39: $\mathcal{A}_p = \mathcal{E}_p = \emptyset$ {nicht mehr zu gebrauchen da p zu Element wird}
40: $k = k + |\mathbf{p}|$
41: **end while**

Lemma 3.5 *In jedem Schritt k des obigen Algorithmus gilt:*

- *Ist die Variable $i \in T \backslash \overline{T}$ kein Repräsentant, dann $\mathcal{A}_i = \mathcal{E}_i = \emptyset$*
- *Ist die Variable $i \in \overline{T}$ ein Repräsentant, dann $\mathcal{A}_i \subseteq T$ und $\mathcal{E}_i \subseteq \overline{S}$*
- *Seien $i, j \in \overline{T}$. Dann gilt entweder $\boldsymbol{j} \subseteq \mathcal{A}_i$ oder $\boldsymbol{j} \cap \mathcal{A}_i = \emptyset$*
- *Sei die Variable $i \in \overline{T}$ ein Repräsentant und $\widetilde{T}_i = \mathcal{A}_i \cap \overline{T}$. Dann stellt die Familie $(\boldsymbol{j})_{j \in \widetilde{T}_i}$ eine Partitionierung von $\mathcal{A}_i$ dar*
- *Da $i \notin \mathcal{A}_i$, folgt $\boldsymbol{i} \cap \mathcal{A}_i = \emptyset$*
- *Ist $e \in S \backslash \overline{S}$ kein Repräsentant (für ein Superelement), dann $\mathcal{L}_e = \emptyset$*
- *Ist $e \in \overline{S}$ ein Repräsentant (für ein Superelement), dann $\mathcal{L}_e \subseteq T$*
- *Seien $i \in \overline{T}$ und $e \in \overline{S}$. Dann gilt entweder $\boldsymbol{i} \subseteq \mathcal{L}_e$ oder $\boldsymbol{i} \cap \mathcal{L}_e = \emptyset$*
- *Sei $e \in \overline{S}$ ein Repräsentant (für ein Superelement) und $\widetilde{T}_e = \mathcal{L}_e \cap \overline{T}$. Dann stellt die Familie $(\boldsymbol{j})_{j \in \widetilde{T}_e}$ eine Partitionierung von $\mathcal{L}_e$ dar*
- *Ist $e \in \mathcal{E}_i$ dann $\boldsymbol{i} \subseteq \mathcal{L}_e$*
- *Ist die Variable $i \in \overline{T}$ ein Repräsentant, dann*

$$\mathcal{A}_i \cap \left(\bigcup_{e \in \mathcal{E}_i} \mathcal{L}_e \right) = \emptyset$$

  Anders formuliert: sind zwei Variablen $i, j \in \overline{T}$ in $\mathcal{G}$ über $\overline{S}$ verbunden, dann gibt es keine direkte Kante zwischen i und j in $\mathcal{G}$. Gab es diese Kante irgenwann früher, so muss sie während des Algorithmus als überflüssig eingestuft und entfernt worden sein

- *Ist die Variable $i \in \overline{T}$ ein Repräsentant, dann*

$$\mathrm{Adj}_{G_k}(\boldsymbol{i}) = \left(\mathcal{A}_i \cup \left(\bigcup_{e \in \mathcal{E}_i} \mathcal{L}_e \right) \right) \backslash \boldsymbol{i}$$

- *Ist die Variable $i \in \overline{T}$ ein Repräsentant, dann ist sein genaue externer Grad:*

$$d_i = |\,\mathrm{Adj}_{G_k}(i) \backslash \boldsymbol{i}| = |\mathcal{A}_i| + \left| \left(\bigcup_{e \in \mathcal{E}_i} \mathcal{L}_e \right) \backslash \boldsymbol{i} \right| = |\mathcal{A}_i| + \left| \bigcup_{e \in \mathcal{E}_i} (\mathcal{L}_e \backslash \boldsymbol{i}) \right|$$

Sieht man das Lemma, so scheint es sinnvoll, bei der Mengen $(\mathcal{A}_k)_k$ und $(\mathcal{L}_e)_e$ nur Repräsentanten i aus $\overline{T}$ zu speichern und nicht das ganze $\mathbf{i}$ (die Mengen $\mathbf{i}$ sollten getrennt anderswo gespeichert werden). Das Problem wäre dann bei der Verschmelzung zwei alter Supervariablen in einer neuen Supervariable (Zeile 29). Man müsste den alten Repräsentanten j aus allen $(\mathcal{A}_k)_{k\in\mathcal{A}_j=\mathcal{A}_i}$ und $(\mathcal{L}_e)_{e\in\mathcal{E}_j=\mathcal{E}_i}$ entfernen.

Den AMD-Algorithmus erhält man, indem man überall d_i durch den Näherungswert $\overline{d}_i$ ersetzt. Ist die Variable p das k-te Pivot und $i \in \mathcal{L}_p \cap \overline{T}$, dann definiere $\overline{d}_i$:

$$\overline{d}_i^{\,k} = \begin{cases} |\mathcal{A}_i^1| = |\{j : \{i,j\} \in E\}| & \text{für } k = 1 \\ \min \left\{ \begin{array}{l} n-k \\ \overline{d}_i^{\,k-1} + |\mathcal{L}_p \backslash \mathbf{i}| \\ |\mathcal{A}_i| + |\mathcal{L}_p \backslash \mathbf{i}| + \sum_{e\in\mathcal{E}_i\backslash\{p\}} |\mathcal{L}_e \backslash \mathcal{L}_p| \end{array} \right\} & \text{für } k \geq 2 \end{cases} \tag{3.2}$$

Jetzt wird es klar wieso das unvollstandige degree update für den obigen Algorithmus nicht gut geeignet ist: um $\overline{d}_i^{\,k}$ zu bestimmen braucht man $\overline{d}_i^{\,k-1}$.

Lemma 3.5 liefert auch die Begründung, wieso der folgende Algorithmus die Größen $|\mathcal{L}_e \backslash \mathcal{L}_p|$ (bei 3.2) berechnet:

Algorithmus 3.6 Berechnung der Größen $|\mathcal{L}_e \backslash \mathcal{L}_p|$ für alle $i \in \mathcal{L}_p \cap \overline{T}$

```
1: vorausgesetzt w(k) < 0 für alle k = 1, 2, ..., n
2: for i ∈ L_p ∩ T̄ do {durchlaufe alle Repräsentanten in L_p}
3:    for e ∈ E_i do {durchlaufe alle (super)Elementen e ∈ E_i}
4:       if w(e) < 0 then
5:          w(e) = |L_e|
6:       end if
7:       w(e) = w(e) − |i| {aus Lemma 3.5}
8:    end for
9: end for
```

3.4 Genauigkeit des approximierten Grades

Gilbert, Moler und Schreiber ([38]) benutzen bei ihren AMD (die Matlab-Routinen `colmmd` und `symmmd`) folgende Approximation $\widehat{d}_i$ für externe Grade:

$$\widehat{d}_i = |\mathcal{A}_i| + \sum_{e\in\mathcal{E}_i} |\mathcal{L}_e \backslash \mathbf{i}|$$

Oft besitzt die Pivotvariable zwei oder weniger adjazente (super)Elemente. Daher haben Ashcraft, Eisenstat und Lucas ([7]) folgende modifizierte Approximation

$\widetilde{d}_i$ vorgeschlagen:

$$\widetilde{d}_i = \begin{cases} d_i & \text{falls } |\mathcal{E}_i| = 2 \\ \widehat{d}_i & \text{sonst} \end{cases}$$

Satz 3.6 ([6]) *Falls* $|\mathcal{E}_i| \leq 1$, *dann* $d_i = \overline{d}_i = \widetilde{d}_i = \widehat{d}_i$. *Falls* $|\mathcal{E}_i| = 2$, *dann* $d_i = \overline{d}_i = \widetilde{d}_i \leq \widehat{d}_i$. *Falls* $|\mathcal{E}_i| > 2$, *dann* $d_i \leq \overline{d}_i \leq \widetilde{d}_i = \widehat{d}_i$. *Also, immer gilt* $d_i \leq \overline{d}_i \leq \widetilde{d}_i \leq \widehat{d}_i$.

3.5 COLAMD und SYMAMD

COLAMD[7] ([24]) liefert eigentlich eine fill-in reduzierende unsymmetrische Permutation. Durch geschickte Anwendung kann man aber eine fill-in reduzierende symmetrische Permutation daraus gewinnen, die SYMAMD. COLAMD ist für LU-Zerlegungen entwickelt worden, die nur Zeilenpivotisierung erlauben.

Sei A eine nicht singuläre $n \times n$ Matrix mit nullfreier Diagonale. Die COLAMD-Autoren zeigen in [24], wie man eine obere Dreieckstruktur $\mathcal{R}_A$ bildet, sodass die Strukturen aller U_P's, welche aus eine LU-Zerlegung mit Zeilenpivotisierung $PA = L_P D_P U_P$ zu gewinnen sind, in ihr enthalten sind. COLAMD liefert eine Permutation $\psi \leftrightarrow Q$, sodass $\mathcal{R}_{AQ^T}$ klein ist. Wie beim MD wird hier eine greedy-Strategie angewandt, denn eine globale Minimierung ist für solche Probleme normalerweise NP-hart. Auch COLAMD benutzt Näherungswerte für seine Metrik.

Warum ist COLAMD auch für symmetrische Matrizen interessant?

Satz 3.7 *Sei* A *eine nichtsinguläre* $n \times n$ *Matrix mit nullfreier Diagonale. Sei* $PA = L_P D_P U_P$ *eine* LU*-Zerlegung mit Zeilenpivotisierung und sei* $A^T A = L_C D L_C^T$. *Dann gilt* $\text{Struct}(L_P) \subseteq \text{Struct}(L_C)$ *und* $\text{Struct}(U_P) \subseteq \text{Struct}\big(L_C^T\big)$.

Beweis: Da A eine nullfreie Diagonale besitzt, gilt $\text{Struct}(A) \subseteq \text{Struct}\big(A^T A\big)$. Auch PA besitzt eine nullfreie Diagonale, so dass $\text{Struct}(PA) \subseteq \text{Struct}\big((PA)^T(PA)\big)$ $= \text{Struct}\big(A^T(P^T P)A\big) = \text{Struct}\big(A^T A\big)$. Daraus folgt $\text{Struct}(L_P) \subseteq \text{Struct}(L_C)$ und $\text{Struct}(U_P) \subseteq \text{Struct}\big(L_C^T\big)$. □

Satz 3.8 ([39]) *Sei* A *eine* $n \times n$ *nichtsinguläre Matrix mit nullfreier Diagonale, so dass* $G(A)$ *stark zusammenhängend ist. Sei* $A^T A = L_C D L_C^T$. *Für jedes* $(i,j) \in \text{Struct}\big(L_C^T\big)$ *gibt es eine Permutationsmatrix* P, *so dass*[8] $PA = L_P D_P U_P$ *mit* $(i,j) \in \text{Struct}(U_P)$.

[7] Column approximate minimum degree

[8] wie immer, zufällige Nullen gibt es nicht

Ist also A regulär mit nullfreier Diagonale und $G(A)$ stark zusammenhängend, dann ist Struct$\left(L_C^T\right)$ die Hülle für die Struktur aller Matrizen U, welche aus einer LU-Zerlegung von A mit Zeilenpivotisierung entstehen.

Sei A eine symmetrische $n \times n$ Matrix (mit nullfreier Diagonale). SYMAMD bildet erst eine Matrix M, sodass Struct$\left(M^T M\right)$ = Struct(A). Für jede Permutationsmatrix $\widetilde{Q}$ sei $\widetilde{Q}A\widetilde{Q}^T = L_{\widetilde{Q}} L_{\widetilde{Q}}^T$. Es gilt

$$\text{Struct}\left(\widetilde{Q}A\widetilde{Q}^T\right) = \text{Struct}\left(\widetilde{Q}M^T M\widetilde{Q}^T\right) = \text{Struct}\left((M\widetilde{Q}^T)^T(M\widetilde{Q}^T)\right).$$

Sei außerdem $PM\widetilde{Q}^T = L_P^{\widetilde{Q}} D_P^{\widetilde{Q}} U_P^{\widetilde{Q}}$. Da Struct$\left(L_{\widetilde{Q}}^T\right)$ eine einigermaßen scharfe[9] Obermenge für die Hülle aller möglichen Struct$\left(U_P^{\widetilde{Q}}\right)$ ist, liefert COLAMD, auf M angewandt, eine Permutationsmatrix Q, für welche die Hülle aller möglichen Struct$\left(U_P^Q\right)$[10] klein ist. Dadurch wird indirekt auch Struct$\left(L_Q^T\right)$ klein sein, d.h. Q wirkt Fill-in-reduzierend auf A. Somit wird die Cholesky-Zerlegung von QAQ^T ein kleineres Fill-in haben. Die Ausgabe von SYMAMD(A) ist Q =COLAMD(M).

3.6 Vergleiche

Als Testmatrizen A dienen (quadratische) unsymmetrische Matrizen. Gibt es Nullen in der (Haupt-) Diagonale von A, so wird eine (fürs erste beliebige) Transversale angewandt. Um die Struktur von A zu symmetrisieren, betrachten wir A als eingebettet in $(A + A^T)$.

In die Tests wird auch die approximierte Mindeg Routine `symmmd` aus MATLAB miteinbezogen.

Folgende Abbildungen 3.5 und 3.6 zeigen die Matrizen *af23560* und *av41092* permutiert durch `amd`, `symamd` und `symmmd`. Informationen über das Fill-in sind auch mitgeliefert.

Folgende Tabelle 3.1 zeigt die Dauer der Permutationsroutinen und das Fill-in, nachdem die Permutationen angewandt wurden. Die Routinen wurden mittels ihres Matlab Interface aufgerufen. Alle drei Routinen sind kompiliert; während `amd` und `symamd` über ihrer mex-file Schnittstellen aufgerufen werden, ist `symmmd` eine "built-in"-Funktion von MATLAB. Ab MATLAB 7, ist sogar `symamd` eine "built-in"-Funktion.

Tests, welche die RCM-Permutation enthalten, sind nicht aufgelistet. Diese Permutation ist den anderen (MinDeg basierten) Permutationen hoffnungslos un-

[9] Unter den Voraussetzungen des Satzes 3.8 ist Struct$\left(L_{\widetilde{Q}}^T\right)$ sogar genau die Hülle.

[10] $PMQ^T = L_P^Q D_P^Q U_P^Q$

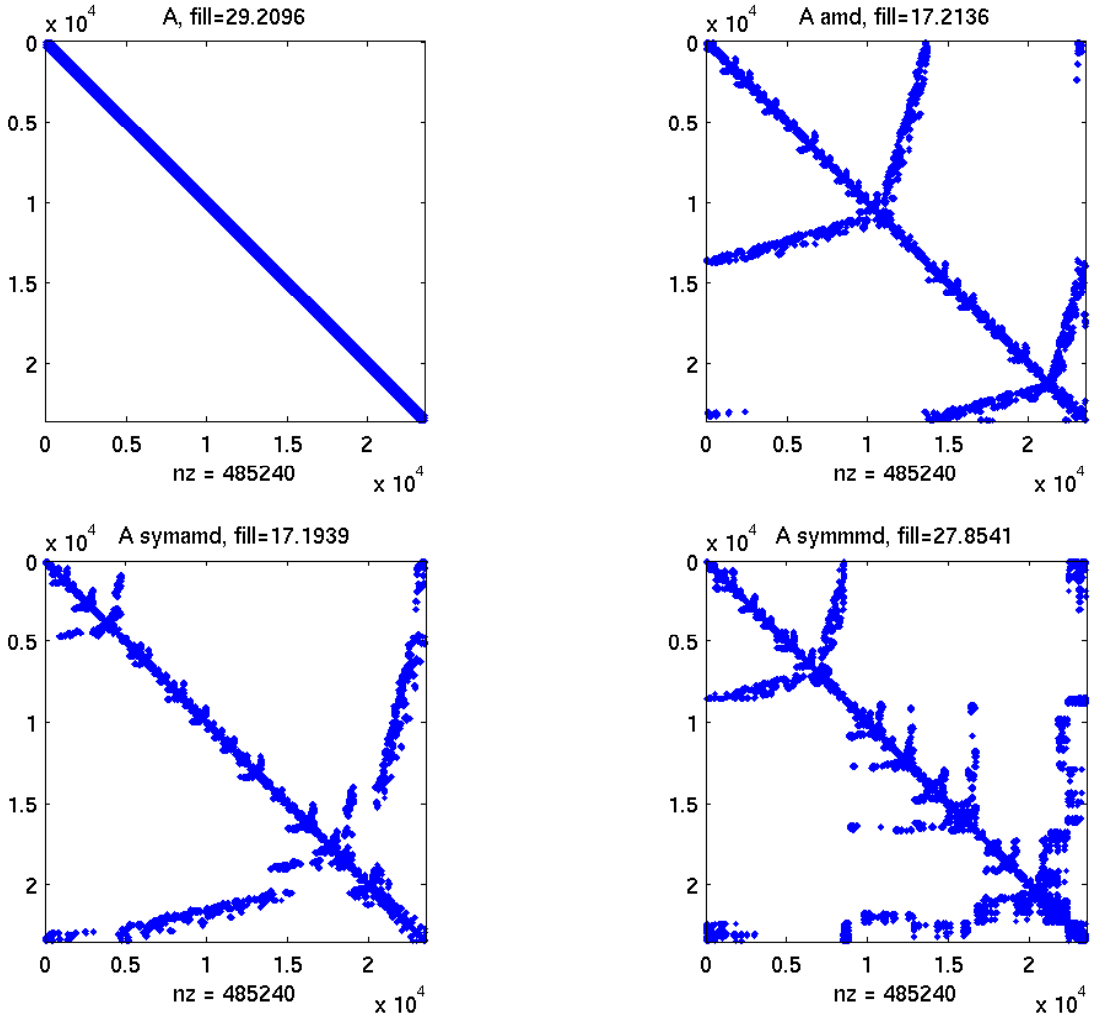


Abbildung 3.5: Matrix *af23560*

terlegen. Testmaschine war eine P4 HT 2.8GHz, Linux Suse 8.2. Alle Zeitangaben sind in Sekunden.

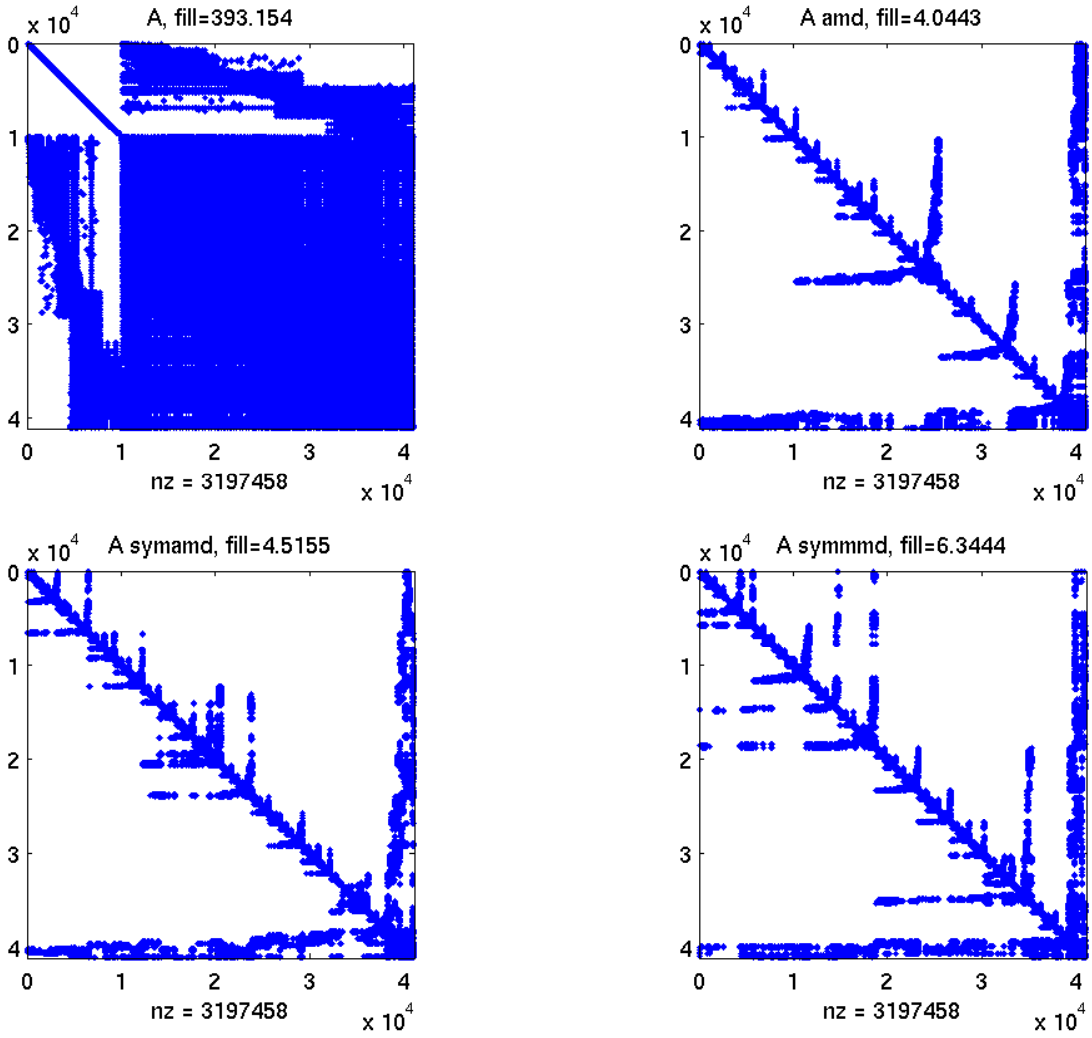
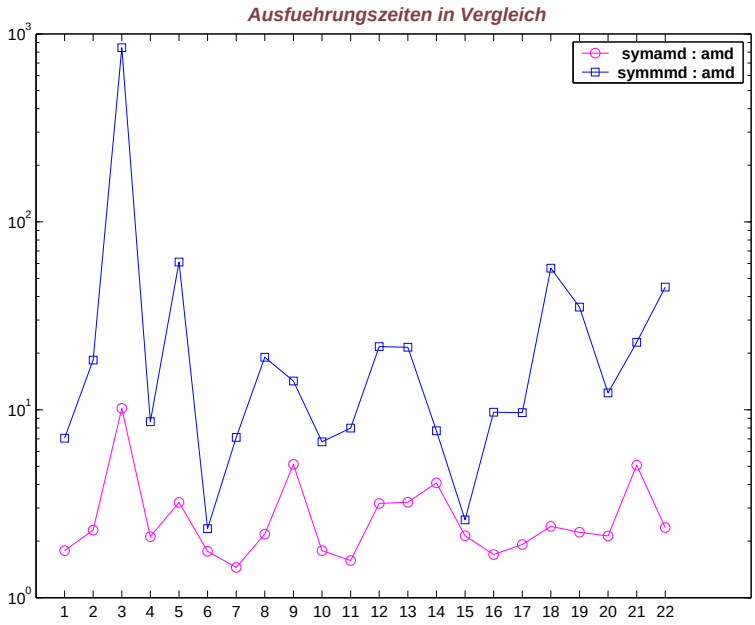


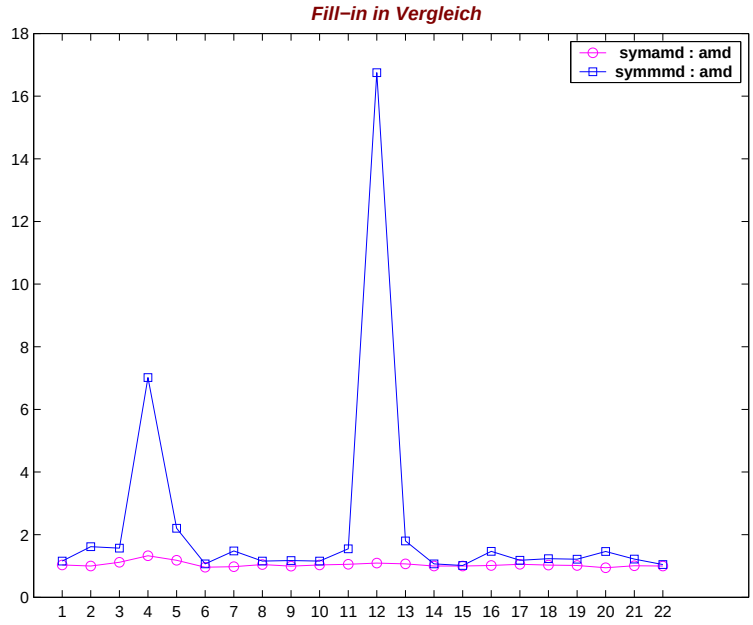
Abbildung 3.6: Matrix *av41092*. Eine Transversale wurde erst angewandt

Matrix Nr.	Matrix Name	Dim	fill-in	amd Dauer	amd fill-in	symamd Dauer	symamd fill-in	symmmd Dauer	symmmd fill-in
1	3D_51448_3D	51448 × 51448	356.9888	0.3710	51.7837	0.6596	53.4865	2.6162	59.9650
2	af23560	23560 × 23560	29.2096	0.1553	17.2136	0.3555	17.1939	2.8549	27.8541
3	av41092	41092 × 41092	393.1540	0.7049	4.0443	7.1702	4.5155	594.9380	6.3444
4	bayer01	57735 × 57735	7.5689	0.2418	5.4847	0.5103	7.2762	2.0884	38.4820
5	bbmat	38744 × 38744	17.4004	0.5430	17.8530	1.7447	21.0769	33.1977	39.3730
6	dw8192	8192 × 8192	135.9182	0.0203	9.6770	0.0358	9.2763	0.0473	10.3077
7	ecl32	51993 × 51993	726.6991	0.3794	101.2860	0.5496	99.1468	2.7047	150.0884
8	g7jac200	59310 × 59310	112.5966	0.7629	27.2351	1.6621	28.4977	14.5094	31.5294
9	goodwin	7320 × 7320	11.0769	0.0598	4.1407	0.3064	4.1163	0.8500	4.8642
10	ibm_matrix_2	51448 × 51448	356.9888	0.3758	51.7837	0.6689	53.4865	2.5357	59.9650
11	jan99ac120	41374 × 41374	190.3856	0.4448	11.5957	0.7015	12.2104	3.5507	17.9632
12	lhr71	70304 × 70304	139.5732	0.7140	6.3256	2.2667	6.9023	15.4727	105.9491
13	li	22695 × 22695	29.4534	0.2979	45.0219	0.9587	47.9360	6.4071	81.0613
14	lung2	109460 × 109460	132.4084	0.0859	1	0.3503	1	0.6640	1.0655
15	memplus	17758 × 17758	2744.6	0.0437	1.2162	0.0934	1.2166	0.1134	1.2373
16	matrix-new_3	125329 × 125329	400.0206	0.6837	55.8483	1.1607	56.7681	6.6299	82.0160
17	matrix_9	103430 × 103430	467.4702	0.7531	146.2361	1.4432	154.1577	7.2702	173.0149
18	onetone1	36057 × 36057	134.2721	0.3065	11.4093	0.7339	11.7315	17.3587	14.0420
19	onetone2	36057 × 36057	82.8406	0.1502	4.8902	0.3347	4.9610	5.2789	5.9454
20	para-9	155924 × 155924	6219.6	1.3658	170.7442	2.9106	161.3622	16.8059	249.9608
21	rim	22560 × 22560	13.0590	0.1852	4.4919	0.9398	4.5276	4.2313	5.4861
22	twotone	120750 × 120750	246.5897	1.1858	10.3195	2.7986	10.3007	53.2568	10.7881

Tabelle 3.1: *amd*, *symamd* und *symmmd* in Vergleich

Der Inhalt dieser Tabelle wird durch die zwei folgenden Plots verdeutlicht. Die Reihenfolge der Matrizen ist identisch mit der in der Tabelle.





Man sieht sofort dass die Permutationsgüte von AMD und SYMAMD miteinander vergleichbar sind, die von SYMMMD ist schlechter. Bezüglich der Ausführungszeiten, ist AMD der deutliche Gewinner. Auf Grund dieser Ergebnisse wurde entschieden, in dieser Arbeit stets auf AMD zurückzugreifen.

Kapitel 4

Die multifrontale Methode

4.1 Der Eliminationsbaum

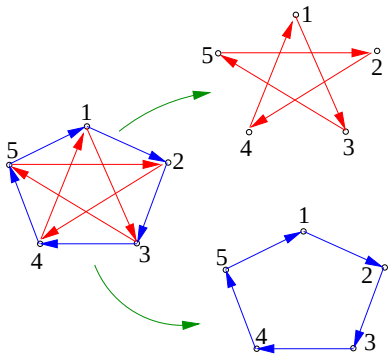
4.1.1 Die transitive Reduktion eines Digraphen

Die transitive Reduktion eines Digraphen ([5]) ist ein effizienter Weg, um Informationen über gerichtete Pfade zu speichern.

Definition 4.1 *Sei $G = (V, E)$ ein Digraph. Ein Digraph $TR(G) = (V, E')$ heißt eine **transitive Reduktion**, falls folgende zwei Bedingungen erfüllt werden:*

1. *$\forall u, v \in V$ gibt es einen gerichteten Pfad von u nach v in $TR(G)$ dann und nur dann, wenn es einen solchen Pfad von u nach v in G gibt.*

2. *Es gibt keinen gerichteten Graphen $G'' = (V, E'')$ mit weniger Kanten, der die erste Eigenschaft besitzt.*



Die Existenz einer transitiven Reduktion ist offensichtlich: Die Menge der Digraphen mit Knotenmenge V ist endlich, und ein Digraph, der die erste Eigenschaft besitzt, ist G selber. Wie das Bild links zeigt, Digraphen die Zyklen enthalten besitzen keine eindeutige transitive Reduktion. Gleich werden wir zeigen, falls der Digraph keinen Zyklus besitzt, d.h. der Digraph ist ein DAG, dann ist die transitive Reduktion eindeutig.

Damit es beim Punkt 1 der Definition 4.1 für jeden gerichteten Pfad von u nach v in G einen gerichteten Pfad von u nach v in $TR(G)$ gibt, ist die Existenz eines gerichteten Pfades von u nach v in $TR(G)$ für alle Kanten $(u,v) \in E$ notwendig und hinreichend.
Der Vorteil der transitiven Reduktion ist, dass sie viel weniger Kanten besitzen kann als der Graph selber, aber alle Informationen über gerichtete Pfade erhalten bleiben.
Wir interessieren uns insbesondere für die transitive Reduktion bei DAGs. Folgende Definition gilt für alle Digraphen.

Definition 4.2 *Ein Digraph $G = (V,E)$ heißt **transitiv**, wenn $\forall u \neq v \in V$ gilt: Gibt es einen gerichteten Pfad von u nach v gibt, dann ist $(u,v) \in E$.*
***Transitive Closure** $TC(G)$ ist der transitive Digraph G' mit $V(G') = V(G)$, so dass $E(G) \subseteq E(G')$ und $|E(G')|$ minimal ist.*

Bei DAGs erhält man die transitive Closure, indem man gerichtete Kanten zwischen verschiedenen Knoten, die mit einem gerichteten Pfad verbunden sind, aber nicht direkt durch eine Kante, hinzufügt.

Satz 4.1 ([5]) *Sei $G = (V,E)$ ein DAG. Es gibt einen eindeutigen Digraph G^t mit der Eigenschaft $TC(G^t) = TC(G)$, und für jeden echten Teilgraph H von G^t gilt $TC(H) \neq TC(G)$. Der Graph G^t ist gegeben durch $G^t = \bigcap_{G' \in S(G)} G'$, wobei*

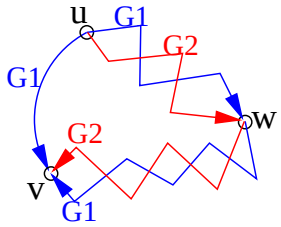
$$S(G) = \{\, G' = (V, E(G')) \mid TC(G') = TC(G) \,\}.$$

Beweis: Der Beweis ergibt sich direkt aus den zwei folgenden Lemmata[1]:

Lemma 4.1 *Seien G_1 und G_2 zwei DAGs, mit $V(G_1) = V(G_2)$ und $TC(G_1) = TC(G_2)$. Falls es eine Kante $(u,v) \in G_1$ mit $(u,v) \notin G_2$ gibt, dann gilt*

$$TC(G_1 - \{(u,v)\}) = TC(G_1) = TC(G_2)$$

Beweis:



Wegen $(u,v) \in G_1$, $(u,v) \notin G_2$ und $TC(G_1) = TC(G_2)$, gibt es einen gerichteten Pfad von u nach v in G_2, der durch einen dritten Knoten w läuft. Wiederum wegen $TC(G_1) = TC(G_2)$, gibt es zwei gerichtete Pfade in G_1, der eine von u nach w, der andere von w nach v. Keiner von beiden Pfaden kann die Kante (u,v) enthalten, weil das einen Zyklus ergeben würde. Damit ist gezeigt, dass die Kante (u,v) in G_1 „umgehbar“ ist, es ist $TC(G_1 - \{(u,v)\}) = TC(G_1)$. □

[1] Im Folgenden, falls $G = (V,E)$ und $\widetilde{E} \subseteq E$, dann werden wir unter $G - \widetilde{E}$ den Graphen $(V, E \backslash \widetilde{E})$ verstehen.

Lemma 4.2 *Sei G ein DAG. Die Menge $S(G) = \{ G' \mid TC(G') = TC(G) \}$ ist abgeschlossen bezüglich Vereinigung und Durchschnitt.*

Beweis: Seien $G_1, G_2 \in S(G)$. Weil $TC(G_1) = TC(G_2) = TC(G)$, gilt $G_1 \bigcup G_2 \subseteq TC(G)$, und weil $TC(G)$ transitiv ist, $TC(G_1 \bigcup G_2) \subseteq TC(TC(G)) = TC(G)$. Andererseits ist $G_1 \subset G_1 \bigcup G_2$, also $TC(G) = TC(G_1) \subseteq TC(G_1 \bigcup G_2))$. Somit gilt $TC(G_1 \bigcup G_2) = TC(G)$, d.h. $(G_1 \bigcup G_2) \in S(G)$.

Sei $\{a_1, a_2, \ldots, a_r\}$ die Kantenmenge der Digraphen $G_1 - (G_1 \cap G_2)$. Wendet man das Lemma 4.1 mehrere Male nacheinader an, so ergibt sich:

$$\begin{aligned} TC(G_1 - \{a_1\}) &= TC(G_1) \\ TC(G_1 - \{a_1\} - \{a_2\}) &= TC(G_1) \\ &\vdots \\ TC(G_1 - \{a_1\} - \{a_2\} - \ldots - \{a_r\}) &= TC(G_1) \end{aligned}$$

Die letzte Gleichung ist nichts anderes als $TC(G_1 \bigcap G_2) = TC(G_1) = TC(G)$, d.h. $(G_1 \bigcap G_2) \in S(G)$. □

□

Der Digraph G^t ist nicht anderes als die transitive Reduktion $TR(G)$ des DAG G. Aus dem Beweis sieht man, dass man um G^t zu erhalten, nur die "umgehbaren" Kanten zu löschen braucht.

4.1.2 Eine Vereinheitlichung aller Gauß-Varianten

Sei A eine $(n \times n)$ Matrix. Sei $M_n = \{1, 2, \ldots, n\} \times \{1, 2, \ldots, n\}$.
Sei $\lhd$ die partielle Ordnung in M_n :

$$\forall (i_1, j_1) \neq (i_2, j_2) \in M_n, ((i_1, j_1) \lhd (i_2, j_2) \Longleftrightarrow (i_1 \leq i_2) \wedge (j_1 \leq j_2)).$$

Es ist offensichtlich zu sehen, dass $(1, 1)$ das kleinste und (n, n) das größte Element in $(M_n, \lhd)$ ist.

Man kann jeder möglichen LDU Gauß-Variante (z.B. ikj, kij, Crout-Doolittle ([30]) usw.) eine Folge (sog. Gaußelimination-Permutation)
$\sigma_G = (\underbrace{(i_1, j_1)}_{(1,1)}, \ldots, \underbrace{(i_{n^2}, j_{n^2})}_{(n,n)})$ aus $(M_n, \lhd)$ zuordnen, die folgende Eigenschaft erfüllt:

$$(\forall t \in 1, \ldots, n^2)(\{(i, j) \in M_n \mid (i, j) \lhd (i_t, j_t)\} \subseteq ((i_1, j_1), \ldots, (i_{t-1}, j_{t-1}))).$$

Dies bedeutet, dass die Elemente, die kleiner als (i_t, j_t) sind, früher vorkommen. Umgekehrt, gehört zu jeder solche Permutation eine LDU Gauß-Variante. Der folgende Algorithmus macht die 1-1 Beziehung zwischen Gauß-Varianten und Gaußelimination-Permutationen σ_G klar:

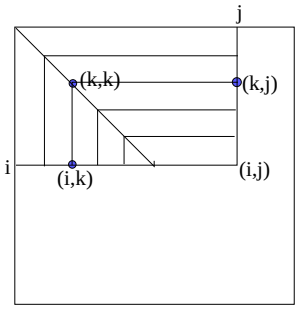


Abbildung 4.1:

Algorithmus 4.1 *LDU* - Gauss all in one

Beschreibung: Input: A, $\sigma_G = ((i_1, j_1), \ldots, (i_{n^2}, j_{n^2}))$ aus $(M_n, \triangleleft)$ wie oben; Output: L,D,U "in place"

1: **for** $t = 1, \ldots, n^2$ **do**

2: Sei $(i, j) = (i_t, j_t)$

3: $\lambda = a_{i,j} - \sum_{k=1}^{\min(i,j)-1} a_{i,k} a_{k,k} a_{k,j}$; {Beachte hier $(i,k), (k,k), (k,j) \triangleleft (i,j)$}

4: $a_{i,j} = \begin{cases} \lambda & i = j \\ \lambda / a_{i,i} & i < j \\ \lambda / a_{j,j} & i > j \end{cases}$

5: **end for**

4.1.3 Eine strukturelle Aussage

Wie immer wird angenommen, dass keine zufälligen Nullen auftreten. Sei A eine $(n \times n)$ Matrix mit nullfreier Diagonale. Sei $A = LDU$ die Gauß-Zerlegung und $F = L+U$ (wir sind nur an der Struktur von F interesiert). $G(F)$ ist der sogenante *fill Graph* von A.

Satz 4.2 *Sei $i \neq j$. Es gilt $(i, j) \in G(F)$ dann und nur dann, wenn es in $G(A)$ einen (gerichteten und einfachen) Weg von i nach j gibt mit Zwischenknoten aus $\{1, \ldots, \min(i, j) - 1\}$, d.h. $j \in \mathrm{Reach}_{G(A)}(i, \{1, \ldots, \min(i, j) - 1\})$. (Wenn $(i, j) \in G(A)$, braucht man gar keine Zwischenknoten).*

Beweis: Durch Induktion nach $(i + j)$ mit $3 \leq (i + j) \leq (2n - 1)$.
Falls $i + j = 3$, $i = 1$,$j = 2$ oder $i = 2$,$j = 1$. Weil $F_{2,1} = A_{2,1}/A_{1,1}$ und $F_{1,2} = A_{1,2}/A_{1,1}$ gilt, dann $(1, 2) \in G(F) \iff (1, 2) \in G(A)$ und $(2, 1) \in G(F) \iff (2, 1) \in G(A)$.
Angenomen, die Behauptung stimmt für alle (i', j') mit $3 \leq i' + j' < m$ (wobei

$m \leq (2n-1)$). Sei jetzt (i,j) beliebig mit $i+j=m$ $(i \neq j)$. Aus Abb. 4.1 erkennt man $((i,j) \in G(F)) \Leftrightarrow [(i,j) \in G(A)] \vee [\exists 1 \leq k < \min(i,j))((i,k) \in G(F) \wedge (k,j) \in G(F)]$.

($\Longrightarrow$) Sei $(i,j) \in G(F)$. Falls $(i,j) \in G(A)$, dann ist alles klar. Sei jetzt $(i,j) \in G(F) \backslash G(A)$. Wähle von allen oben erwähnten k's das größte. Aus der Induktionsvoraussetzung gibt es in $G(A)$ einen Weg von i nach k mit Zwischenknoten aus $\{1,\ldots,k-1\}$ und einen Weg von k nach j mit Zwischenknoten aus $\{1,\ldots,k-1\}$. Verbindet man diese Wege (durch k), so erhält man den gesuchten Weg.

($\Longleftarrow$) Umgekehrt, sei $i,k_1,\ldots,k_s,j$ ein (gerichteter) Weg, sodass alle $k_1,\ldots,k_s$ kleiner als i und j sind. Ist $s=0$, so ist $(i,j) \in G(A) \subseteq G(F)$. Sei jetzt $s \geq 1$ und $k = \max\{k_1,\ldots,k_s\}$. Aus der Induktionsvoraussetzung folgt $(i,k) \in G(F)$ und $(k,j) \in G(F)$. Da $k<i$ und $k<j$, gilt $(i,j) \in G(F)$. □

Korollar 4.1 *Es gilt*

$$\begin{aligned}\text{Struct}(U_{i,i+1:n}) &= \text{Reach}_{G(A)}(i,\{1,\ldots,i-1\})\\ \text{Struct}(L_{j+1:n,j}) &= \text{Reach}_{G(A^T)}(j,\{1,\ldots,j-1\})\end{aligned}$$

Ist A (wenigstens strukturell) symmetrisch, dann gilt

$$\text{Struct}(L_{j+1:n,j}) = \text{Reach}_{G(A)}(j,\{1,\ldots,j-1\})$$

4.1.4 SPD-Fall: Cholesky Zerlegung

Wenn A symmetrisch ist, dann ist $U = L^T$. In diesem Fall ist es logisch, den Gauß-Algorithmus zu vereinfachen. Unter allen möglichen Gauß-Varianten, gilt die unten aufgeführte Spalten-Variante als die beste. Sie ist im Grunde eine Variante des Crout-Doolittle Algorithmus für symmetrische Matrizen:

Algorithmus 4.2 Cholesky Zerlegung: $A = LDL^T$

Beschreibung: Input: A; Output L, D

1: **for** $i = 1,\ldots,n$ **do** {berechne D_i und Spalte $L_{:,i}$}

2: $$\begin{pmatrix} t_i \\ \vdots \\ t_n \end{pmatrix} = \begin{pmatrix} A_{i,i} \\ \vdots \\ A_{n,i} \end{pmatrix} - \sum_{k<i} L_{i,k} D_k \begin{pmatrix} L_{i,k} \\ \vdots \\ L_{n,k} \end{pmatrix}$$

3: $$D_i = t_i, \quad \begin{pmatrix} L_{i,i} \\ \vdots \\ L_{n,i} \end{pmatrix} = \frac{1}{t_i} \begin{pmatrix} t_i \\ \vdots \\ t_n \end{pmatrix}$$

4: **end for**

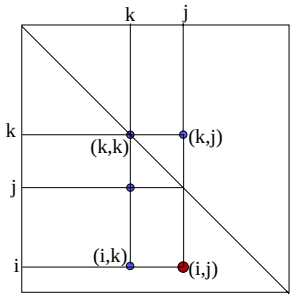
4.1.5 Der Eliminationsbaum

Der Eliminationsbaum wird erst für irreduzible SPD-Matrizen definiert. Die Erweiterung auf nicht irreduzible oder nicht SPD-Matrizen wird offensichtlich sein.

Sei A eine $(n \times n)$ SPD-Matrix. Sei $A = LDL^T$ ihre Cholesky-Zerlegung. Während $G(L)$ und $G(L^T) = G(L)^T$ DAGs sind, ist $G(L + L^T)$ ungerichtet. Sei ab jetzt $F = L + L^T$.
Wichtig ist die sog. *Sehnen*-Eigenschaft bei $G(F)$: Jeder Zyklus der Länge ≥ 4 besitzt eine Sehne. Solche Graphen heißen auch Chordal-Graphen oder Perfect-Elimination-Graphen.

Satz 4.3 *Seien i, j, k drei paarweise verschiedene Knoten, sodass $k < i, j$ und $\{i, k\}, \{j, k\} \in G(F)$. Dann ist auch $\{i, j\} \in G(F)$.*

Beweis: Da $F_{i,k} \cdot F_{k,k} \cdot F_{k,j} \neq 0$, sieht man aus dem Algorithmus 4.1, dass $F_{i,j} \neq 0$.

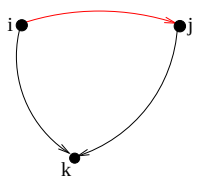


□

Die *Sehnen*-Eigenschaft bei $G(F)$ erhält man, indem man in einem Zyklus (der Länge ≥ 4) den Knoten mit dem kleinsten Index wählt (das soll k sein, i und j die Nachbarn) und den Satz 4.3 anwendet.

Man kann jetzt diese Eigenschaft auf $G(L)$ und $G(L^T)$ übertragen:

Korollar 4.2 *Seien i, j, k drei paarweise verschiedene Knoten, sodass $k < j < i$ und $(i, k), (j, k) \in G(L)$. Dann ist auch $(i, j) \in G(L)$. Analoges gilt für $G(L^T)$.*



Satz 4.4 *Sei A symmetrisch und irreduzibel. Dann gilt*

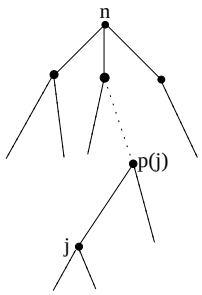
$$(\forall j = 1, \ldots, n-1)(\text{Struct}(L_{j+1:n,j}) \neq \emptyset)$$

Beweis: Es ist vorausgesetzt, dass $G(A)$ zusamenhängend ist. Sei $j_0 = j, j_1, \ldots, j_t$ der kürzeste Weg in $G(A)$, welcher j und $\{j+1, \ldots, n\}$ verbindet. Wegen der Minimalitäts-Eigenschaft, muss $\{j_1, \ldots, j_{t-1}\} \subseteq \{1, \ldots, j-1\}$ gelten, also $j_t \in \text{Reach}_{G(A)}(j, \{1, \ldots, j-1\})$. Aus Korollar 4.1 folgt, dass $\text{Struct}(L_{j+1:n,j}) = \\ = \text{Reach}_{G(A)}(j, \{1, \ldots, j-1\}) \ni j_t$. □

Jetzt sind wir bereit, den Begriff des Eliminationsbaums einzuführen. Sei

$$(\forall 1 \leq j < n),\ p(j) = \min(\text{Struct}(L_{j+1:n,j})) \in \{j+1, j+2, \ldots, n\} \tag{4.1}$$

Definition 4.3 ([46]) *Der Eliminationsbaum $T(A)$ ist der Baum mit Wurzel n, und für jeden anderen Knoten $j \in \{1, 2, \ldots, n-1\}$ sei $p(j)$ sein Vaterknoten.*



Vom Kontext her wird immer klar sein, ob wir $T(A)$ als ungerichtet oder gerichtet (mit Kantenrichtung $(p(j), j)$) ansehen. Hauptsächlich werden wir es mit dem ungerichteten Baum zu tun haben.

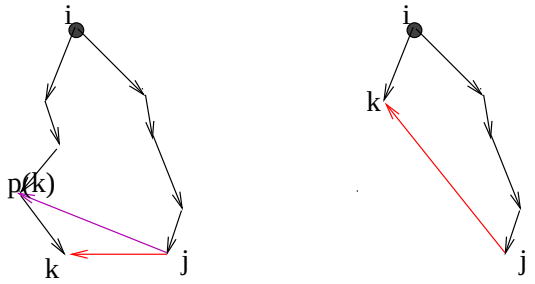
Es bezeichne $T[i]$ den Teilbaum von $T(A)$ mit Wurzel i.

Satz 4.5 ([46]) *Als gerichteter Baum betrachtet, ist $T(A)$ die transitive Reduktion des DAG $G(L)$.*

Beweis: Die Kanten von $T(A)$ sind $E(T(A)) = \{(p(j), j) \mid j = 1, \ldots, n-1\}$. Da $L_{p(j),j} \neq 0$ gilt $E(T(A)) \subseteq E(G(L))$, und weil $T(A)$ ein Baum ist, gibt es keine „umgehbaren“ Kanten. Sei $(j, k) \in G(L)$.

Es bleibt nur zu zeigen, dass $k \in T[j]$. Der Knoten k kann in $T[A]$ nicht ein Vorgänger für j sein, weil $k < j$. Angenomen $k \notin T[j]$. Dann müssen $T[k]$ und $T[j]$ knotendisjunkte Teilbäume sein. Sei i ihr erster gemeinsamer Vorfahre. Zwischen allen solchen Kanten $(j, k) \in G(L)$, sodass $T[k]$ und $T[j]$ knotendisjunk

sind, wähle diejenige mit dem kleinsten Abstand zwischen k und j in $T(A)$ (als ungerichtet betrachtet). Aus $(j,k) \in G(L)$ und Gleichung 4.1 folgt $j \geq p(k)$. Da $k \in T[p(k)]$, kann $j = p(k)$ unmöglich sein. Also, $j > p(k)$. Aus $(j,k) \in G(L)$, $(p(k),k) \in G(L)$ und Korollar 4.2, ergibt sich $(j,p(k)) \in G(L)$. Wären $T[p(k)]$ und $T[j]$ knotendisjunkt, dann widerspräche dies der Minimalitäts-Eigenschaft des Paares (j,k). Die einzig übrig gebliebene Möglichkeit ist $p(k) = i$. Der Knoten i ist ein echtes Vorfahre von j. Dadurch würde gelten $p(k) = i > j$ (Widerspruch!).

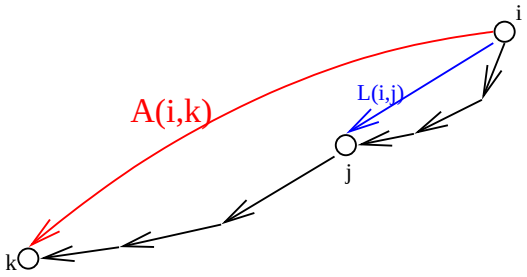


□

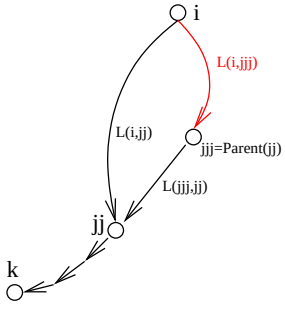
Korollar 4.3 *Falls $L_{i,j} \neq 0$, dann $j \in T[i]$.*

Der folgende Satz ist von fundamentaler Bedeutung:

Satz 4.6 ([46]) *Sei $i > j$. Es gilt: $L_{i,j} \neq 0 \iff (\exists k \in T[j])(A_{i,k} \neq 0)$.*

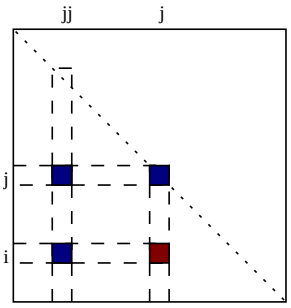


Beweis:



($\Longleftarrow$) Sei $k \in T[j]$ mit $A_{i,k} \neq 0$. Es gilt $i > j > k$. $(A_{i,k} \neq 0) \Rightarrow (L_{i,k} \neq 0) \Rightarrow (k \in T[i])$. Wir haben also $(k \in T[i])$, $(k \in T[j])$ und $i > j$. Da $T[A]$ ein Baum ist, muss $j \in T[i]$ gelten. Es wird jetzt gezeigt (Induktion), dass für alle Knoten jj in dem Pfad von k nach i, $L_{i,jj} \neq 0$. Für $jj = k$ stimmt dies. Angenommen, $L_{i,jj} \neq 0$ für einen Knoten jj in dem Pfad von k nach i, wobei $k \leq jj < i$. Sei $jjj = Parent(jj)$. Da $(i,jj),(jjj,jj) \in G(L)$, folgt

aus Korollar 4.2 dass $(i, jjj) \in G(L)$.



($\Longrightarrow$) Sei $i > j$ mit $L_{i,j} \neq 0$. Wir zeigen mit Induktion über $j \in \{1, \ldots, i-1\}$, dass es ein $k \in T[j]$ mit $A_{i,k} \neq 0$ gibt. Für $j = 1$ ist offensichtlich $(k = 1 = j \in T[j])$. Angenommen die Behauptung stimmt für alle $jj = 1, \ldots, j-1$. Zu zeigen ist die Behauptung für j $(< i)$. Falls $A_{i,j} \neq 0$, dann nehme $k = j \in T[j]$. Sei $A_{i,j} = 0$. Aus dem Cholesky-Algorithmus (Seite 69), gibt es $jj \in \text{Struct}(L_{j,1:j-1})$ mit $L_{i,jj} \neq 0$ und daher $jj \in T[j] \subset T[i]$ (Korollar 4.3). Wendet man die Induktionsvoraussetzung auf jj an, so gibt es ein $k \in T[jj]$ mit $A_{i,k} \neq 0$. Außerdem, gilt $k \in T[jj] \subset T[j]$. Damit ist die Behauptung auch für j bewiesen.

□

Korollar 4.4 *Sei $i > k$ und $A_{i,k} \neq 0$. Für alle Knoten j, die sich in $T[A]$ auf dem Pfad von i nach k (einschließlich i und k) befinden, gilt $L_{i,j} \neq 0$. Alle Nichtnull Einträge der i-ten Zeile $L_{i,:}$ erhält man auf diese Art und Weise.*

Man erhält also die i-te Zeile von L durch Abschneiden aller Teilbaüme des Baums $T[i]$, die Nichtnullen von A *nicht* enthalten. Anders gesagt, entferne aus $T[i]$ aller Teilbaüme $T[j] \subset T[i]$ mit $(\forall k \in T[j])$, $(j, k) \notin \text{Struct}(A)$. Es bezeichne $T_r[i]$ diesen abgeschnittenen Teilbaum ($Zeile(i)$-Teilbaum). Diese Tatsache bildet die Grundlage für den Algorithmus, welches $T[A]$ berechnet.

Bemerkung 4.1 *Ein Knoten i ist ein Blattknoten bzgl. $T(A)$ genau dann, wenn* $\text{Struct}(A_{i,1:i-1}) = \emptyset$.

Interessant ist es, dass man den Eliminationsbaum $T[A]$ (d.h. den Array *Parent*) berechnen kann, ohne dafür erst die Struktur von L zu kennen. Folgender Algorithmus benutzt Wurzelbäume, um disjunkte Mengen darzustellen und die „Path-Compression"-Heuristik ([64],[21]) bei der Operation $find()$ (bekannte Standardtechnik). Dort wird ein Array *next* benutzt, um die Wurzelbäume zu implementieren. Dies ist so zu verstehen:

$$next(i) = \begin{cases} 0 & i \text{ gehört (noch) zu keine Menge} \\ i & i \text{ gehört zu der Menge mit Repräsentant } i \\ sonst & \left\{ \begin{array}{c} i \text{ gehört zu der Menge mit Repräsentant} \\ find(i) = next(...(next(i)...)) \end{array} \right\} \end{cases}$$

Algorithmus 4.3 Berechne $T[A]$(nach [46])

Beschreibung: Input: A; Output $Parent$

```
1:  Parent(1 : n) = 0
2:  next(1 : n) = 0;
3:  for i = 1 : n do {Zeilen von A}
4:     next(i) = i
5:     for k ∈ Struct(A_{i,1:i−1}) do
6:        u = find(k); {der Repräsentant der Menge, zu dem k gehört}
7:        if (Parent(u) == 0) ∧ (u ≠ i) then
8:           Parent(u) = i; {u ist eine echte Sohn von i in T[A]}
9:           next(u) = i; {vereinige Mengen mit Repr. i und u. Neuer Repr. ist i}
10:       end if
11:    end for
12: end for
```

Satz 4.7 ([46]) *Der obige Algorithmus, implementiert mit Path Compression, berechnet $T[A]$ in einer durchnittlichen Laufzeit von $\mathcal{O}(nnz(A)\log_2(n))$.*

Kennt man $T[A]$, so kann man einiges über L einfach und schnell berechnen. So berechnet z.B. der folgender Algorithmus die Anzahl der Nichtnullen in jeder Zeile von L.

Algorithmus 4.4 nnz für jeder Zeile von L

Beschreibung: Input: A, $Parent$; Output $\eta(i) = nnz(L_{i,:})$

```
1:  Setze η(1 : n) = 1 {Die Diagonalelementen}
2:  Setze marker(1 : n) = 0 {Markiert als nicht besucht}
3:  for i = 2, . . . , n do {Zeilen von L}
4:     marker(i) = i
5:     for k ∈ Struct(A_{i,1:i−1}) do
6:        j = k {Von hier aus, steige empor in T(A) Richtung i}
7:        while marker(j) ≠ i do
8:           η(i) + +
9:           marker(j) = i
10:          j = Parent(j)
11:       end while
12:    end for
13: end for
```

Der Aufwand beträgt offensichtlich $\mathcal{O}(nnz(L))$. Somit kann man schnell das Fill-in berechnen. Man kann den Algorithmus ergänzen, um daraus die Anzahl der Nichtnullen in jeder Spalte und auch die Struktur von L zu berechnen (in $\mathcal{O}(nnz(L))$). Es folgen jetzt einige Aussagen über die Struktur der Spalten von L. Die Beweise beruhen auf Satz 4.6.

Satz 4.8 *Für jeden Knoten j, ist der induzierte Teilgraph $G(A)\Big|_{V(T[j])}$ zusammenhängend.*

Satz 4.9 $\text{Struct}(L_{:,j}) = \{i | L_{i,j} \neq 0\} = \{j\} \bigcup Adj_{G(A)}(T[j])$ *und alle ihre Elemente liegen in dem $T[A]$-Pfad, welcher j und die Wurzel n verbindet.*

Satz 4.10 $\text{Struct}(L_{:,j}) = \text{Struct}(A_{j:n,j}) \bigcup \left\{ \bigcup_{p(k)=j} [\text{Struct}(L_{:,k}) - \{k\}] \right\}$.

Jetzt sehen wir, wie man effizient die (sparse) Cholesky-Zerlegung (siehe Algorithmus, Seite 69) implementiert. Aus Zeile 2 entnimmt man, dass nur die Spalten $L_{:,k}$ mit $k \in (T_r[i] - \{i\})$ relevant für die Berechnung der Spalte $L_{:,i}$ sind. Dabei liefert Satz 4.10 deren Struktur.

Algorithmus 4.5 Effiziente Cholesky Zerlegung: $A = LDL^T$

Beschreibung: Input: A; Output L, D

1: Berechne $T[A]$ (d.h. Array *Parent*)
2: **for** $i = 1, \ldots, n$ **do**
3: Berechne $S = \text{Struct}(L_{:,i}) = \{i_0 = i < i_1 < \ldots < i_s\}$ mittels Satz 4.10
4: Setze $L_{S,i} = A_{S,i}$
5: **for** $k \in (T_r[i] - \{i\})$ **do**
6: $L_{S,i} = L_{S,i} - D_k L_{i,k} L_{S,k}$; {Beachte, dass $\text{Struct}(L_{i:n,k}) \subseteq S$}
7: **end for**
8: $D_i = L_{i,i}$, $L_{S,i} = L_{S,i} \,/\, L_{i,i}$
9: **end for**

Korollar 4.5 *Spalte $L_{:,i}$ hängt vom Spalte $L_{:,j} \iff j \in T[i]$. Falls $T[i]$ und $T[j]$ knotendisjunkte Teilbäume sind, dann kann man die Spalten $L_{:,i}$ und $L_{:,j}$ parallel berechnen.*

Nun zu den Fällen, wo $G(A)$ nicht zusammenhängend oder A unsymmetrisch ist.

1. Falls A symmetrisch und $G(A)$ nicht zusamenhängend ist, existiert eine Permutation π, so das $P_\pi A P_\pi^T$ eine (symmetrische) Blockdiagonalmatrix ist, wobei jeder Block zusammenhängend ist (Kapitel 2). Man bestimmt dann für jeden Diagonalblock den Eliminationsbaum.

2. Falls A unsymmetrisch ist, existieren zwei Permutationen π, ψ, so dass $P_\pi A P_\psi^T$ untere Blockdreiecksmatrix, mit nullfreier Diagonale ist (Kapitel 2). Jeder der Diagonalblöcke B ist stark zusamenhängend. Somit sind alle $(B + B^T)$ symmetrisch, $G(B + B^T)$ zusamenhängend und man betrachtet die Matrix B als eingebettet in die $\text{Struct}\big(B + B^T\big)$. Man bestimmt dann für jeden Diagonalblock den Eliminationsbaum.

4.2 Topologische Anordnungen

Sei A symmetrisch.

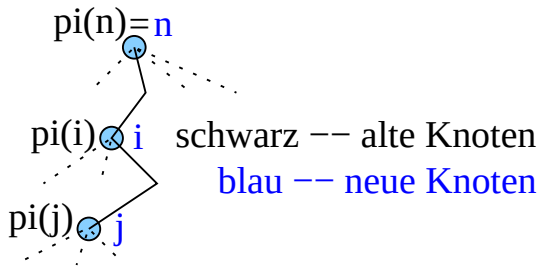
Definition 4.4 *Die Permutation π heißt fill-äquivalent, falls die* fill Graphen *von A und $P_\pi A P_\pi^T$ isomorph sind.*

Basierend auf dem Eliminationsbaum kann man einige fill-äquivalente Permutationen konstruieren.

Definition 4.5 *Sei T ein Wurzelbaum. Eine topologische Anordnung bezüglich T ist eine Anordnung, bei der die Söhne immer vor ihren Eltern numeriert werden.*

Satz 4.11 *Sei A symmetrisch und $G(A)$ zusammenhängend. Jede topologische Anordnung bezüglich $T[A]$ ist eine fill-äquivalente Permutation.*

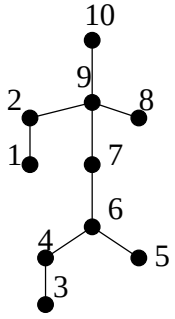
Beweis:



Sei Knoten $\pi(k)$ als k-ter numeriert.
Sei $A = L_A D_A L_A^T$. Dann ist
$B = P_\pi A P_\pi^T = P_\pi L_A D_A L_A^T P_\pi^T =$
$= P_\pi L_A P_\pi^T P_\pi D_A P_\pi^T P_\pi L_A^T P_\pi^T =$
$= \underbrace{\left(P_\pi L_A P_\pi^T\right)}_{L_B} \underbrace{\left(P_\pi D_A P_\pi^T\right)}_{D_B} \underbrace{\left(P_\pi L_A P_\pi^T\right)^T}_{L_B^T}.$

Es ist klar, dass L_B Einheitsdiagonale besitzt und D_B Diagonalmatrix ist. Es wird noch gezeigt, dass L_B untere Dreieckmatrix ist. Falls $L_B(i,j) = L_A(\pi(i), \pi(j)) \neq 0$, dann ist $\pi(j) \in T[\pi(i)]$, d.h. $\pi(i)$ ist ein Vorfahre von $\pi(j)$. Da π topologisch ist, muss $i > j$ sein.
Somit wurde bewiesen, dass $L_B D_B L_B^T$ die Cholesky-Zerlegung von B ist. Also ist $G(L_A) \overset{\pi}{\cong} G(L_B)$ (und daraus: $T[A] = TR(G(L_A)) \overset{\pi}{\cong} TR(G(L_B)) = T[B]$). □

Typische topologische Anordnungen bezüglich $T[A]$ sind Postorderings:



Definition 4.6 *Eine Anordnung bezüglich $T[A]$ heißt Postordering, falls die Knoten in jedem Teilbaum $T[i], i = 1, \ldots, n$, konsekutiv numeriert werden, wobei die Wurzel i immer als letztes numeriert wird.*
Solche Anordnungen findet man als *DFS*-Numerierung ([21]) in Wurzelbäumen. Bei der multifrontalen Methode ist die Anwendung von Postorderings sehr wichtig.

4.3 Die Multifrontale Methode

4.3.1 Die frontale Methode

Die Anfänge der frontalen Methode ([33],[30]) liegen in der Systemlösung von Finite-Elemente-Problemen. Dort entstehen Systeme $Ax = b$ mit A als "Summe" z.B. von (3×3) Cliquen (durch Gebietstriangulierung). Im Folgenden werden wir das Problem allgemein behandeln, völlig abgekoppelt von finiten Elementen.

Sei jetzt A eine sparse $n \times n$ Matrix welche die Gestalt:

$$A = \sum_{e=1}^{nel} B^{[e]} \tag{4.2}$$

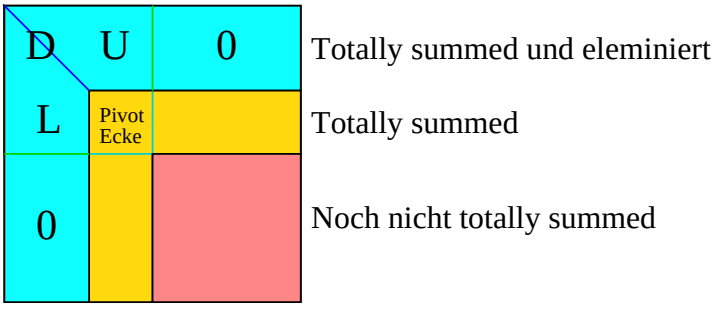
besitzt. *nel* (number of elements) muss nicht von Größenordnung $\mathcal{O}(1)$ sein, sie kann auch $\mathcal{O}(n)$ sein. Seien $\mathcal{S}^{[e]} = \text{Struct}\left(B^{[e]}\right)$, $\mathcal{R}^{[e]} = \{i : \text{Struct}\left(B^{[e]}_{i,:}\right) \neq \emptyset\}$ die Zeilenvariablen und $\mathcal{C}^{[e]} = \{j : \text{Struct}\left(B^{[e]}_{:,j}\right) \neq \emptyset\}$ die Spaltenvariablen des e-ten Elements. Es gilt $\mathcal{S}^{[e]} \subseteq \mathcal{R}^{[e]} \times \mathcal{C}^{[e]}$. Die Menge $\mathcal{V}^{[e]} = \mathcal{R}^{[e]} \cup \mathcal{C}^{[e]}$ sind die Variablen, die beim Element e erscheinen. Die Matrizen $B^{[e]}$ werden im sparse Format gespeichert. Jede elementare Operation

$$a_{i,j} := a_{i,j} + b^{[e]}_{i,j} \quad \text{mit } (i,j) \in \mathcal{S}^{[e]} \tag{4.3}$$

heißt **assembly**. Ein Eintrag (i,j) heißt **fully summed**, wenn alle elementaren Operationen 4.3 durchgeführt sind. Es gibt auch die elementaren Eliminationsschritte

$$a_{i,j} := a_{i,j} - a_{i,k} a^{-1}_{k,k} a_{k,j} \tag{4.4}$$

Der Term $(-a_{i,k} a^{-1}_{k,k} a_{k,j})$ ist ein sogenannter **Rank-1 Beitrag**.
Die frontale Methode beruht auf der Tatsache, dass man Schritt 4.4 durchführen kann, ohne dass der Eintrag (i,j) fully summed ist. Nur die Einträge $a_{i,k}$, $a_{k,k}$ und $a_{k,j}$ müssen **totally summed** sein, d.h. fully summed und es wurden für sie alle Rank-1 Beiträge aus früheren Eliminationsschritten bereits durchgeführt. Damit man einen Pivotknoten ganz 'los wird', d.h. eliminiert, muss seine Spalte und Zeile totally summed sein. Das gibt einen großen Freiheitsraum in der Gestaltung der Gauß-Elimination.



Algorithmus 4.6 Die **frontale** Methode mit Pivotisierung, grob beschrieben

Beschreibung: Input: $A = \sum_{e \in E} B^{[e]}$, Output $PAQ^T = LDU$

1: $F = zeros(n)$, $k = 0$
2: $\pi = \psi = 1 : n$ {Einheitspermutationen}
3: $FS\mathcal{R} = \emptyset$; {Bereits fully summed Zeilen}
4: $FS\mathcal{C} = \emptyset$; {Bereits fully summed Spalten}
5: **while** $E \neq \emptyset$ **do**
6: Entferne eine Element e aus E; {Sollte nicht dem Zufall überlassen sein.}
7: Führe assembly Operationen $f_{i,j} := f_{i,j} + b_{i,j}^{[e]}$ aus für alle $(i,j) \in \mathcal{S}^{[e]}$
8: Sei FSR die Menge der dadurch neu entstandene totally summed Zeilen; $FS\mathcal{R} = FS\mathcal{R} \bigcup FSR$
9: Sei FSC die Menge der dadurch neu entstandene totally summed Spalten; $FS\mathcal{C} = FS\mathcal{C} \bigcup FSC$
10: **while** Es gibt ein 'akzeptables' Pivot $fsr \in FS\mathcal{R}$ und $fsc \in FS\mathcal{C}$ **do**
11: $k = k + 1$
12: permutiere Zeile fsr und Spalte fsc in die obere linke Ecke des Pivot Blocks (das entspricht die Position $F(k,k)$)
13: datiere π und ψ auf
14: $FS\mathcal{R} = FS\mathcal{R} - \{fsr\}$, $FS\mathcal{C} = FS\mathcal{C} - \{fsc\}$
15: Führe alle Operationen $f_{i,j} := f_{i,j} - f_{i,k} f_{k,k}^{-1} f_{k,j}$ aus (Rank-1 Update, dadurch bleibt die totally summed Eigenschaft in $FS\mathcal{R}$ und $FS\mathcal{C}$ erhalten)
16: Berechne $L_{:,k}$, D_k, $U_{k,:}$
17: **end while**
18: **end while**
19: **if** $k < n$ **then** {es ist ein $(n-k)\times(n-k)$ Block unfaktorisiert übrig geblieben}
20: Faktorisiere es nach eine frei wählbare Methode
21: **end if**

4.3.2 Die multifrontale Methode

Die Reihenfolge $(e_1, \ldots, e_{|E|})$ in welcher die einzelnen Elemente e aus E entfernt wurden, entspricht einem assembly Prozess nach der Klammerung:

$$\left(\ldots \left(\left(B^{[e_1]} \right) + B^{[e_2]} \right) \ldots + B^{[e_{|E|}]} \right), \tag{4.5}$$

wobei jedes Klammerpaar (wenigstens) einen Eliminationsversuch darstellt, mit oder ohne Pivotisierung.

Man kann aber auf verschiedene Art und Weise klammern. Z.B. für $|E| = 11$,

$$((B^{[1]} + B^{[3]}) + (B^{[2]} + (B^{[6]} + B^{[11]})) + (((B^{[4]}) + B^{[8]} + B^{[9]}) + ((B^{[5]}) + B^{[7]})) + B^{[10]}) \tag{4.6}$$

Die Klammerung kann am besten durch einen Baum, den sogenannten **assembly tree** dargestellt werden.

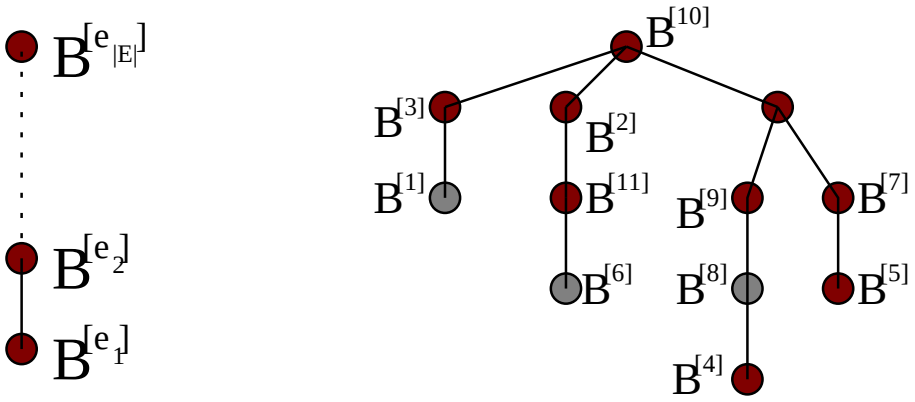
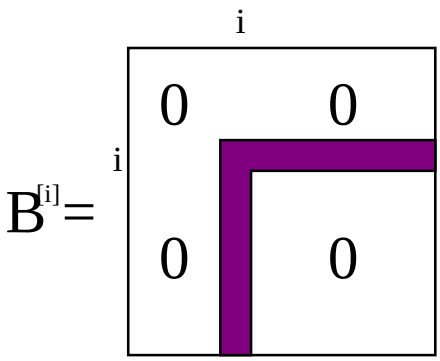


Abbildung 4.2: assembly tree Beispiele für Klammerungen (4.5) und (4.6)

Rote Knoten stehen für eine Klammerpaar, d.h. auch für (wenigstens) einen Eliminationsversuch nach dem assembly Prozess. Es sei angemerkt, dass in Abbildung 4.2 die Knoten $B^{[1]}$ und $B^{[3]}$, $B^{[6]}$ und $B^{[11]}$ sowie $B^{[8]}$ und $B^{[9]}$ vertauscht werden können (die Knotenfarben müssen unangetastet bleiben), weil sie 'den gleichen Rang' innerhalb der Klammerung besitzen.

Natürlich sollte eine Klammerung auf sinnvolle Art vorgenommen werden. Eine wichtige Eigenschaft einer Klammerung ist, wenn man in verschiedenen Zweigen des assembly tree gleichzeitig arbeiten kann. In diesem Fall spricht man von der **multifrontalen Methode**, wenn man an mehreren Fronten gleichzeitig arbeitet. Das ist ein natürlicher Ansatz zur Parallelisierung.

4.3.3 Der Eliminationsbaum als universale assembly tree



Sei A (strukturell) symmetrisch und $G(A)$ zusamenhängend. Das Klammerungsproblem in der multifrontalen Methode ist äquivalent mit der Suche nach geeigneten assembly trees. Überraschenderweise ist der Eliminationsbaum ein guter assembly tree ([47]). Die Matrix A wird in 'Winkeln' partitioniert. Sei $E = \{1, \ldots, n\}$. Die Elementmatrizen $(B^{[i]})_{i \in E}$ definiert man wie folgt:

$$B^{[i]}_{i,i} = A_{i,i},\ B^{[i]}_{i,i+1:n} = A_{i,i+1:n},\ B^{[i]}_{i+1:n,i} = A_{i+1:n,i},\ \text{sonst } B^{[i]}_{k,j} = 0$$

d.h. nur der i-te Winkel von A wird in $B^{[i]}$ übernommen, sonst 0.

Alle Knoten sollen **rot** sein und die Elementmatrix $B^{[j]}$ steht am Knoten j.

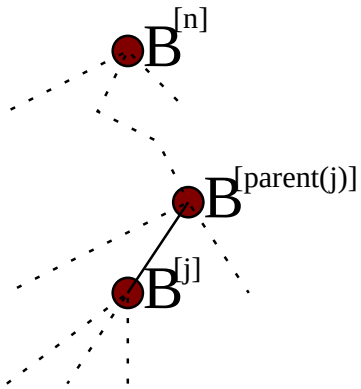


Abbildung 4.3: Assembly tree aus dem Eliminationsbaum

Im Folgenden wird beschrieben, wie man die Multifrontale Methode effizient implementieren kann (ab jetzt dient immer $T[A]$ als assembly tree).

Es wird angenommen, dass keine inakzeptablen Pivots auftreten. Sei $A = LDU$. Da A strukturell symmetrisch ist, $\text{Struct}(L) = \text{Struct}\left(U^T\right)$ und strukturell gesehen kann die Faktorisierung als eine Cholesky-Zerlegung betrachtet werden. Für $j = 1, \ldots, n$ definiere $\mathcal{L}_j^* = \text{Struct}(L_{j:n,j}) = \{j_0 = j < j_1 < \ldots < j_{r(j)}\}$ und $\mathcal{L}_j = \mathcal{L}_j^* - \{j\}$.
Des Weiteren, definiere als **Beiträgematrix des j-te Teilbaums** die $(r(j) + 1) \times (r(j) + 1)$ Matrix:

$$\widetilde{U}_j = - \sum_{k \in T[j] - \{j\}} L_{\mathcal{L}_j^*, k} D_k U_{k, \mathcal{L}_j^*} \tag{4.7}$$

(U wie Update, nicht zu verwechseln mit dem U aus $A = LDU$). Nach Satz 4.10 (notfalls rekursiv angewendet) ist gesichert, dass $\text{Struct}(L_{j:n,k}) \subseteq \mathcal{L}_j^* = \text{Struct}(L_{j:n,j})$. Das bedeutet, dass $\widetilde{U}_j$ alle Rank-1 Updates zu den Pivots, welche echte Nachfahren von j, sind, enthält.

Definiere als **j-te frontale Matrix** die $(r(j) + 1) \times (r(j) + 1)$ Matrix:

$$F_j = B^{[i]}_{\mathcal{L}_j^*, \mathcal{L}_j^*} + \widetilde{U}_j = \begin{pmatrix} A_{j,j} & A_{j,\mathcal{L}_j} \\ A_{\mathcal{L}_j,j} & 0 \end{pmatrix} + \widetilde{U}_j \tag{4.8}$$

Beachte hier, dass $\text{Struct}(A_{j+1:n,j}) \subseteq \mathcal{L}_j$.

Man kann unterscheiden, ob ein $k \in T[j] - \{j\}$ (also $k < j$) die Beziehungen

- $k \in T_r[j]$, d.h. $L_{j,k} \neq 0$ (äq. $U_{k,j} \neq 0$) oder
- $k \notin T_r[j]$, d.h. $L_{j,k} = 0$ (äq. $U_{k,j} = 0$)

erfüllt. Daher kann man die Summe (4.7) umschreiben als

$$\widetilde{U}_j = - \sum_{k \in T_r[j]-\{j\}} L_{\mathcal{L}_j^*,k} D_k U_{k,\mathcal{L}_j^*} - \sum_{k \in T[j]-T_r[j]} \begin{pmatrix} 0 \\ L_{\mathcal{L}_j,k} \end{pmatrix} D_k \begin{pmatrix} 0 & U_{k,\mathcal{L}_j} \end{pmatrix} \tag{4.9}$$

Ihre erste Spalte: $\widetilde{U}_j(:,1) = - \sum_{\substack{k<j \\ U_{k,j} \neq 0}} L_{\mathcal{L}_j^*,k} D_k U_{k,j}$ ist der Rank-1 Beitrag, aus früheren Eliminationsschriten, auf die j-te Spalte von L. Ähnliches gilt auch für U. Die erste Spalte und Zeile von F_j sind also totally summed. Nach einem Eliminationsschritt gilt also

$$F_j = \begin{pmatrix} 1 & 0 \\ L_{\mathcal{L}_j,j} & I \end{pmatrix} \begin{pmatrix} D_j & 0 \\ 0 & U_j \end{pmatrix} \begin{pmatrix} 1 & U_{j,\mathcal{L}_j} \\ 0 & I \end{pmatrix} \tag{4.10}$$

mit den **vollen** $(r(j) \times r(j))$ Matrix U_j. Sie heißt **die j-te update Matrix**.

Satz 4.12 ([47]) *Es gilt*

$$U_j = - \sum_{k \in T[j]} L_{\mathcal{L}_j,k} D_k U_{k,\mathcal{L}_j} \tag{4.11}$$

Beweis:

$$F_j = \begin{pmatrix} 1 \\ L_{\mathcal{L}_j,j} \end{pmatrix} D_j \begin{pmatrix} 1 & U_{j,\mathcal{L}_j} \end{pmatrix} + \begin{pmatrix} 0 & 0 \\ 0 & U_j \end{pmatrix} =$$
$$= \begin{pmatrix} A_{j,j} & A_{j,\mathcal{L}_j} \\ A_{\mathcal{L}_j,j} & 0 \end{pmatrix} - \sum_{k \in T[j]-\{j\}} \begin{pmatrix} L_{j,k} \\ L_{\mathcal{L}_j,k} \end{pmatrix} D_k \begin{pmatrix} U_{k,j} & U_{k,\mathcal{L}_j} \end{pmatrix}$$

Lässt man bei allen hier erscheinenden Matrizen die erste Zeile und Spalte weg, so erhält man:

$$L_{\mathcal{L}_j,j} D_j U_{j,\mathcal{L}_j} + U_j = - \sum_{k \in T[j]-\{j\}} L_{\mathcal{L}_j,k} D_k U_{k,\mathcal{L}_j}$$
$$\Longleftrightarrow$$
$$U_j = - \sum_{k \in T[j]} L_{\mathcal{L}_j,k} D_k U_{k,\mathcal{L}_j}$$

□

Um die Assembly-Operationen in sparse-mode durchzuführen, wird die **erweiterte Addition** $\oplus$ eingeführt. Diese Operation ist intuitiv trivial, aber

aufwändig formell zu beschrieben: Seien $S \to (m_S \times n_S)$ und $T \to (m_T \times n_T)$ zwei Matrizen in Cliquen-Format mit dazugehörigen Indexmengen $\mathcal{R}_S = \{\rho_1^S < \ldots < \rho_{m_S}^S\}$, $\mathcal{C}_S = \{\kappa_1^S < \ldots < \kappa_{n_S}^S\}$, $\mathcal{R}_T = \{\rho_1^T < \ldots < \rho_{m_T}^T\}$ und $\mathcal{C}_T = \{\kappa_1^T < \ldots < \kappa_{n_T}^T\}$. Seien $\mathcal{R}_W = \mathcal{R}_S \cup \mathcal{R}_T = \{\rho_1^W < \ldots < \rho_{m_W}^W\}$ mit $m_W = |\mathcal{R}_W|$, $\mathcal{C}_W = \mathcal{C}_S \cup \mathcal{C}_T = \{\kappa_1^W < \ldots < \kappa_{n_W}^W\}$ mit $n_W = |\mathcal{C}_W|$. Seien S_{ind}^{row}, S_{ind}^{col}, T_{ind}^{row}, T_{ind}^{col}, so dass[2] $\mathcal{R}_W(S_{ind}^{row}) = \mathcal{R}_S$, $\mathcal{R}_W(S_{ind}^{col}) = \mathcal{C}_S$, $\mathcal{R}_W(T_{ind}^{row}) = \mathcal{R}_T$, $\mathcal{R}_W(T_{ind}^{col}) = \mathcal{R}_T$. Erweitere nun S zu $\widetilde{S}$ durch Nullen, so dass $\mathcal{R}_{\widetilde{S}} = \mathcal{R}_W$, $\mathcal{C}_{\widetilde{S}} = \mathcal{C}_W$ und $\widetilde{S}(S_{ind}^{row}, S_{ind}^{col}) = S$. Analog wird T zu $\widetilde{T}$ erweitert. Definiere jetzt:

$$W = S \oplus T = \widetilde{S} + \widetilde{T}, \text{ mit Indexmengen } \mathcal{R}_W \text{ und } \mathcal{C}_W$$

Beispiel: $S = \begin{array}{c} \\ 4 \\ 6 \end{array} \begin{array}{c} \begin{array}{cc} 3 & 7 \end{array} \\ \begin{pmatrix} a & b \\ c & d \end{pmatrix} \end{array}$ und $T = \begin{array}{c} \\ 4 \\ 8 \end{array} \begin{array}{c} \begin{array}{cc} 1 & 3 \end{array} \\ \begin{pmatrix} e & f \\ g & h \end{pmatrix} \end{array}$. Dann ist $S \oplus T$:

$$S \oplus T = \begin{array}{c} \\ 4 \\ 6 \\ 8 \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 3 & 7 \end{array} \\ \begin{pmatrix} e & a+f & b \\ 0 & c & d \\ g & h & 0 \end{pmatrix} \end{array}.$$

Es gibt eine enge Beziehung zwischen F_j und U_j:

Satz 4.13 ([47]) *Seien $c_1, \ldots, c_s$ die Kinder des Knotens j im Eliminationsbaum. Dann gilt:*

$$F_j = \begin{pmatrix} A_{j,j} & A_{j,\mathcal{L}_j} \\ A_{\mathcal{L}_j,j} & 0 \end{pmatrix} \oplus U_{c_1} \oplus \ldots \oplus U_{c_s}$$

Beweis: $T[j] - \{j\} = T[c_1] \cup \ldots \cup T[c_s]$. Aus (4.7) ergibt sich

$$\widetilde{U}_j = -\sum_{k \in T[j]-\{j\}} L_{\mathcal{L}_j^*,k} D_k U_{k,\mathcal{L}_j^*} = \sum_{v=1}^{s} \left(-\sum_{k \in T[c_v]} L_{\mathcal{L}_j^*,k} D_k U_{k,\mathcal{L}_j^*} \right).$$

Für $k \in T[c_v]$ ist $\text{Struct}\left(L_{\mathcal{L}_j^*,k}\right) = \text{Struct}(L_{j:n,k}) \subseteq \mathcal{L}_{c_v}$. Daher ist $\widetilde{U}_j$ erhältlich als:

$$\bigoplus_{v=1}^{s} \left(-\sum_{k \in T[c_v]} L_{\mathcal{L}_{c_v},k} D_k U_{k,\mathcal{L}_{c_v}} \right) \underbrace{=}_{\text{Satz 4.12}} \bigoplus_{v=1}^{s} U_{c_v}$$

evt. nach Erweiterung mit Nullen. Im Ergebnis erhalten wir

$$F_j = \begin{pmatrix} A_{j,j} & A_{j,\mathcal{L}_j} \\ A_{\mathcal{L}_j,j} & 0 \end{pmatrix} \oplus \bigoplus_{v=1}^{s} U_{c_v}$$

□

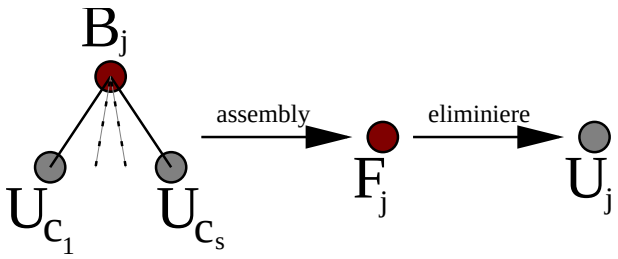
[2] MATLAB-Notation

Satz 4.13 ist die Grundlage der multifrontalen Methode:

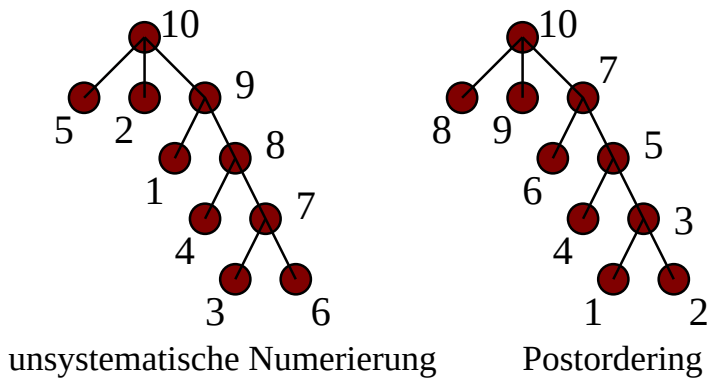
Algorithmus 4.7 $A = LDU$ Zerlegung mit der multifrontalen Methode

Beschreibung: Input: A mit nullfreier Diagonale, strukturell symmetrisch und $G(A)$ zusamenhängend ; Output L, D, U

1: Berechne $T[A]$
2: **for** $j = 1 : n$ **do**
3: Seien $c_1, \ldots, c_s$ die Söhne des Knoten j in $T[A]$
4: Berechne $\mathcal{L}_j^* = \{j_0 = j < j_1 < \ldots < j_{r(j)}\}$ mittels Satz 4.10
5: Berechne $F_j = \begin{pmatrix} A_{j,j} & A_{j,\mathcal{L}_j} \\ A_{\mathcal{L}_j,j} & 0 \end{pmatrix} \oplus U_{c_1} \oplus \ldots \oplus U_{c_s}$ (assembly)
6: Zerstöre die update Matrizen $(U_{c_v})_{v=1,\ldots,s}$ (Speicher freigeben)
7: Führe einen Eliminationsschritt $F_j = \begin{pmatrix} 1 & 0 \\ L_{\mathcal{L}_j,j} & I \end{pmatrix} \begin{pmatrix} D_j & 0 \\ 0 & U_j \end{pmatrix} \begin{pmatrix} 1 & U_{j,\mathcal{L}_j} \\ 0 & I \end{pmatrix}$ aus
8: Bewahre U_j (zusammen mit $\mathcal{L}_j$) für spätere assemby Operationen auf
9: **end for**



Ein Problem bei der Implementierung dieses Algorithmus ist die Verwaltung der Matrizen U_j. Geht man unsystematisch vor, so kann es sein, dass eine Menge Speicher verschwendet wird, wie die folgende Abbildung zeigt. Hier erweisen sich die Postorderings (siehe Definition 4.6) als nützlich.



In [47] wird ein Kriterium für ein bestes Postordering bezüglich des Speicherverbrauchs angegeben. In der Praxis reicht aber ein einfaches Postordering.

Eine wichtige Bemerkung ist, dass man hier Pivotisierung einführen kann, sog. delayed pivots. Die habe ich in meinem ILU Präkonditionierer benutzt. Es wird vorgegangen genau wie bei der frontalen Methode. Ein ausführliches Beispiel für die multifrontale Methode (mit Pivotisierung) wird in Abschnitt 7.1 gegeben.

Die Assembly-Phase (Zeile 5) trägt einen erheblichen Teil an der Ausführungszeiten der direkten multifrontalen Methode. Um dies zu reduzieren, gibt es eine Variante welche **Supernodes** ([47],[49]) verwendet. Die Assembly-Phase bereitet in meinem Programm keine großen Schwierigkeiten, denn ich habe dieses Problem anders beseitigt (Seite 147). Daher wird die Supernode-Variante nicht näher vorgestellt.

Kapitel 5

Iterative Methoden und Präkonditionierung

5.1 Iterative Krylov-Unterraum-Methoden

Da ich nur GMRES bei meinem Code benutzt habe, werde ich nur iterative Methoden, die auf Krylov-Unterräumen basieren, beschreiben.

5.1.1 Allgemeine Projektionsmethode

Sei $Ax = b$ ein LGS, wobei $A \in \mathbb{R}^{n\times n}$ regulär, $x, b \in \mathbb{R}^n$ und x^* die exakte Lösung ist. Sei x_0 eine Anfangsapproximation für x^* und $\mathcal{K}$ und $\mathcal{L}$ zwei m-dimensionale Unterräume von $\mathbb{R}^n$. Die allgemeine Projektionsmethode sucht den nächsten Approximationvektor $\widetilde{x}$ nach der **Petrov-Galerkin Regel** ([58]):

$$\text{Finde } \widetilde{x} \in x_0 + \mathcal{K}, \text{ sodass } b - A\widetilde{x} \perp \mathcal{L}\,. \tag{5.1}$$

Bezeichnet man $\widetilde{x} = x_0 + \delta$ und $r_0 = b - Ax_0$, dann ist $b - A\widetilde{x} = b - A(x_0 + \delta) = r_0 - A\delta$, und man kann die Bedingungen (5.1) wie folgt umschreiben:

$$\widetilde{x} = x_0 + \delta, \quad \delta \in \mathcal{K}\,, \tag{5.2}$$

$$(r_0 - A\delta, w) = 0, \quad \forall w \in \mathcal{L}\,. \tag{5.3}$$

Sei $V = [v_1, \ldots, v_m] \in \mathbb{R}^{n\times m}$, sodass die Spaltenvektoren $v_1, \ldots, v_m$ eine Basis von $\mathcal{K}$ bilden und $W = [w_1, \ldots, w_m] \in \mathbb{R}^{n\times m}$, sodass die Spaltenvektoren

$w_1, \ldots, w_m$ eine Basis von $\mathcal{L}$ bilden. Die Approximierte $\widetilde{x}$ kann man als:

$$\widetilde{x} = x_0 + Vy, \quad \text{mit } y \in \mathbb{R}^m,$$

schreiben. Die Orthogonalitätsbedingung 5.3 ist äquivalent zu:

$$W^T AVy = W^T r_0 \,. \tag{5.4}$$

Ist die $m \times m$ Matrix $W^T AV$ regulär, dann ist die neue Approximierte:

$$\widetilde{x} = x_0 + V(W^T AV)^{-1} W^T r_0 \tag{5.5}$$

Die Matrix $W^T AV$ muss nicht regulär sein. Dann hat (5.1) keine Lösung. In zwei Fällen ist sie aber immer regulär:

Satz 5.1 ([58]) *Gilt eine der folgenden Bedingungen:*

- *A ist positiv definit und $\mathcal{L} = \mathcal{K}$,*
- *A ist regulär und $\mathcal{L} = A\mathcal{K}$,*

dann ist die Matrix $W^T AV$ regulär, unabhängig von der Basisauswahl V für $\mathcal{K}$ und W für $\mathcal{L}$.

Besondere Bedeutung haben die zwei folgenden Aussagen ([58]):

Satz 5.2 *Sei A SPD und $\mathcal{L} = \mathcal{K}$. $\widetilde{x}$ ist die Approximation nach der Petrov-Galerkin Bedingung (5.1) genau dann, wenn sie die A-Norm[1] des Fehlers minimiert:*

$$\|\widetilde{x} - x^*\|_A = \min_{x \in x_0 + \mathcal{K}} \|x - x^*\|_A \,.$$

Da $\widetilde{x}, x_0 \in x_0 + \mathcal{K}$, wird der Lösungsfehler (bezüglich A-Norm) mit Sicherheit nicht vergrößert $\|\widetilde{x} - x^*\|_A \leq \|x_0 - x^*\|_A$.

Satz 5.3 *Sei A regulär und $\mathcal{L} = A\mathcal{K}$. $\widetilde{x}$ ist die Approximation nach der Petrov-Galerkin Bedingung (5.1) genau dann, wenn sie die Norm des Residuums minimiert:*

$$\|b - A\widetilde{x}\|_2 = \min_{x \in x_0 + \mathcal{K}} \|b - Ax\|_2 \,.$$

Da $\widetilde{x}, x_0 \in x_0 + \mathcal{K}$, wird das Residuum (bezüglich 2-Norm) mit Sicherheit nicht vergrößert $\|b - A\widetilde{x}\|_2 \leq \|b - Ax_0\|_2$.

Es folgt ein Algorithmus ([58]), welches die allgemeine Projektionsmethode beschreibt:

[1] $\|z\|_A = \|z^T Az\|_2$

Algorithmus 5.1 Allgemeine Projektionsmethode

Beschreibung: Input: A, m und Altapprox. x_0; Output: Neuapprox. x_m

1: Wähle zwei m-dimensionale Unterräume $\mathcal{K}$ und $\mathcal{L}$
2: Wähle die Basis $V = [v_1, \ldots, v_m]$ für $\mathcal{K}$ und $W = [w_1, \ldots, w_m]$ für $\mathcal{L}$
3: $r_0 = b - Ax_0$
4: $y = (W^T AV)^{-1} W^T r_0$
5: $x_m = x_0 + Vy$

Um noch genauere Näherungswerte für die exakte Lösung x^* zu erhalten, verwendet man **restarted**-Varianten, d.h. starte den Algorithmus neu, evtl. mehrere Mals nacheinander, mit Anfangsnäherung $x_0 = x_m$.

5.1.2 Krylov-Unterraum Methoden

Die Aussagen der Sätze 5.2 und 5.3 bilden die Basis für viele iterative Krylov-Unterraum Methoden.

Sei $Ax = b$ eine LGS, wobei $A \in \mathbb{R}^{n \times n}$ regulär, $x, b \in \mathbb{R}^n$ und x^* die exakte Lösung ist. Sei x_0 eine Anfangsapproximation für x^* und $r_0 = b - Ax_0$.

Sei $m \in \mathbb{N}$, $m \ll n$. Wähle für $\mathcal{K}$

$$\mathcal{K}_m = \mathcal{K}_m(A, r_0) = span\{r_0, Ar_0, A^2 r_0, \ldots, A^{m-1} r_0\}$$

der sog. **Krylov Unterraum**. Trivialerweise gilt

Satz 5.4

$$\mathcal{K}_m(A, r_0) = \{p(A)r_0 \,:\, (p \in \mathbb{R}[x])(deg(p) < m)\}$$

Des Weiteren wird eine orthonormale Basis $V_m = [v_1, \ldots, v_m]$ für $\mathcal{K}_m$ benötigt. Dafür verwendet man das Arnoldi Verfahren mit modifiziertem Gram-Schmidt.

Lemma 5.1 *Mit den Größen aus Algorithmus 5.2 sei* $V_m = [v_1, \ldots, v_m] \in \mathbb{R}^{n \times m}$, $V_{m+1} = [v_1, \ldots, v_m, v_{m+1}] \in \mathbb{R}^{n \times (m+1)}$, $\overline{H}_m = (h_{i,j}) \in \mathbb{R}^{(m+1) \times m}$ *und* $H_m = \overline{H}_m(1 : m, :) \in \mathbb{R}^{m \times m}$. *Dann gelten:*

$$AV_m = V_m H_m + w_m e_m^T = V_{m+1} \overline{H}_m \tag{5.6}$$

$$V_m^T AV_m = H_m \tag{5.7}$$

Wir stellen nun einige konkrete iterative Krylov-Unterraum-Methoden vor.

Algorithmus 5.2 Arnoldi Verfahren mit modifizierten Gram-Schmidt

Beschreibung: Input: A, m und v_1 normalisiert; Output $v_1, \ldots, v_m, v_{m+1}$ orthonormale Basis für $\mathcal{K}_m$, $\overline{H}_m = (h_{i,j}) \in \mathbb{R}^{(m+1)\times m}$ mit $h_{i,j} = 0$ für $i > j+1$ d.h. (obere) Hessenberg-Matrix

1: **for** $j = 1, 2, \ldots, m$ **do**
2: berechne $w_j = Av_j$
3: **for** $i = 1, \ldots, j$ **do**
4: $h_{i,j} = (w_j, v_i)$
5: $w_j = w_j - h_{i,j}v_i$ {Orthogonalisierung gegenüber v_i}
6: **end for**
7: $h_{j+1,j} = \|w_j\|_2$. Falls $h_{j+1,j} = 0$ Stop!
8: $v_{j+1} = w_j/h_{j+1,j}$ {Normalisierung}
9: **end for**

5.1.3 FOM

Sei $A \in \mathbb{R}^{n\times n}$ beliebig aber regulär. Wir nehmen $\mathcal{L} = \mathcal{K} = \mathcal{K}_m(A, r_0)$. Als v_1 für das Arnoldi-Verfahren wählt man $v_1 = r_0/\|r_0\|_2$. Die Iterierte $\widetilde{x}$ wird im folgenden Algorithmus als x_m bezeichnet ([58]).

Algorithmus 5.3 Full Orthogonalisation Method (FOM)

Beschreibung: Input: A, b, x_0 m; Output x_m

1: bestimme $r_0 = b - Ax_0$, $\beta = \|r_0\|$, $v_1 = r_0/\beta$
2: Arnoldi(A, m, v_1) liefert V_m und H_m
3: bestimme $y_m = H_m^{-1}(\beta e_1)$ und $x_m = x_0 + V_m y_m$

5.1.4 GMRES

Sei $A \in \mathbb{R}^{n\times n}$ beliebig aber regulär. Hier wird $\mathcal{K} = \mathcal{K}_m(A, r_0)$ und $\mathcal{L} = A\mathcal{K}$ gewählt. Als v_1 für Arnoldi wählt man $v_1 = r_0/\|r_0\|_2$ ([58],[61]).

Algorithmus 5.4 Generalized Minimum Residual Method (GMRES)

Beschreibung: Input: A, b, x_0 m; Output x_m

1: bestimme $r_0 = b - Ax_0$, $\beta = \|r_0\|$, $v_1 = r_0/\beta$
2: Arnoldi(A, m, v_1) liefert V_m und $\overline{H}_m$
3: bestimme y_m, welches $\left(\|\beta e_1 - \overline{H}_m y\|_2\right)_{y\in\mathbb{R}^m}$ minimiert
4: bestimme $x_m = x_0 + V_m y_m$

Da $\overline{H}_m$ Hessenberg Matrix ist, ist das Problem in Zeile 3 durch die Berechnung der QR-Faktorisierung mit m Givens-Rotationen einfach zu lösen.

Um noch genauere Näherungswerte für die exakte Lösung x^* zu erhalten, verwendet man "restarted GMRES".

Satz 5.5 ([58]) *Ist A positiv definit ist, so ist die restarted GMRES(m) konvergent.*

Satz 5.6 ([58]) *Ist A diagonalisierbar und liegen alle Eigenwerte von A in einer Ellipse mit Achsen parallel zum Koordinatensystem, sodass der Nullpunkt ausserhalb der Ellipse liegt, so ist restarted GMRES(m) konvergent für m groß genug.*

5.1.5 Lanczos: Arnoldi für symmetrische Matrizen

Falls A symmetrisch ist, folgt aus (5.7):

$$H_m^T = V_m^T A^T V_m = V_m^T A V_m = H_m$$

D.h. H_m ist eine symmetrische tridiagonale Matrix. In diesem Fall schreibt man statt H_m

$$T_m = \begin{pmatrix} \alpha_1 & \beta_2 & & & \\ \beta_2 & \alpha_2 & \beta_3 & & \\ & \cdot & \cdot & \cdot & \\ & & \beta_{m-1} & \alpha_{m-1} & \beta_m \\ & & & \beta_m & \alpha_m \end{pmatrix}$$

Dadurch lässt sich der Arnoldi-Algorithmus vereinfachen:

Algorithmus 5.5 Lanczos-Verfahren

Beschreibung: Input: A, m und v_1 normalisiert; Output: orthonormale Vektoren $v_1, \ldots, v_m, v_{m+1}$ und die Einträge $\alpha_1, \ldots, \alpha_m$, $\beta_2, \ldots, \beta_m, \beta_{m+1}$ für die Tridiagonalmatrix T_m

1: setze $\beta_1 = 0$, $v_0 = 0$
2: **for** $j = 1, \ldots, m$ **do**
3: berechne $w_j = Av_j - \beta_j v_{j-1}$
4: $\alpha_j = (w_j, v_j)$ {entspricht $h_{j,j}$}
5: $w_j = w_j - \alpha_j v_j$
6: $\beta_{j+1} = \|w_j\|_2$. Falls $\beta_{j+1} = 0$ Stop! {β_{j+1} entspricht $h_{j+1,j}$}
7: $v_{j+1} = w_j / \beta_{j+1}$
8: **end for**

5.1.6 CG

Sei jetzt A SPD. Wir nehmen $\mathcal{L} = \mathcal{K} = \mathcal{K}_m(A, r_0)$. Als v_1 für Lanczos wählt man $v_1 = r_0/\|r_0\|_2$. FOM vereinfacht sich nun zu ([58])

Algorithmus 5.6 Lanczos Method for Linear Systems

Beschreibung: Input: A, b, x_0 m; Output x_m

1: bestimme $r_0 = b - Ax_0$, $\beta = \|r_0\|$, $v_1 = r_0/\beta$
2: Lanczos(A, m, v_1) liefert V_m und T_m
3: bestimme $y_m = T_m^{-1}(\beta e_1)$ und $x_m = x_0 + V_m y_m$

Durch eine sehr geschickte, überhaupt nicht triviale Modifizierung ([58]), erhält man den CG-Algorithmus, bei welchem nur noch kurze Rekursionen auftreten und von m unabhängiger Speicherbedarf auftritt

Algorithmus 5.7 Conjugate Gradient(CG)

Beschreibung: Input: A, b, x_0 m; Output x_m

1: bestimme $r_0 = b - Ax_0$, $p_0 = r_0$
2: **for** $j = 0, \ldots, m-1$ **do**
3: $\alpha_j = (r_j, r_j)/(Ap_j, p_j)$
4: $x_{j+1} = x_j + \alpha_j p_j$
5: $r_{j+1} = r_j - \alpha_j A p_j$
6: $\beta_j = (r_{j+1}, r_{j+1})/(r_j, r_j)$
7: $p_{j+1} = r_{j+1} + \beta_j p_j$
8: **end for**

CG ist mathematisch äquivalent mit FOM für symmetrisch positiv definite Matrizen.

5.1.7 CR und MINRES

Sei A symmetrisch. Hier wird $\mathcal{K} = \mathcal{K}_m(A, r_0)$ und $\mathcal{L} = A\mathcal{K}$ gewählt. Als v_1 für Arnoldi wählt man $v_1 = r_0/\|r_0\|_2$. Man kann auch hier die Symmetrie von A benutzen, um das GMRES-Verfahren in einen effizienten, CG-ähnlichen Algorithmus umzuformulieren[2] (evt. nächste Seite):

Wie beim CG, basiert der CR-Algorithmus auf der schrittweise "erweiterbaren" LU-Zerlegung der Matrix T_m aus dem Lanczos-Verfahren ([58]). Falls A positiv indefinit ist, kann es zu Null-Pivots kommen und die Folge wäre ein break-down. Dies wird wiedergespiegelt durch Zeilen 3 bei CG und 6 bei CR. Benutzt man aber eine schrittweise "erweiterbare" QR-Zerlegung, dann erhält man den sog. MINRES-Algorithmus ([53]). Hier gibt es keine break-down Gefahr. Sowohl CR als auch MINRES sind mathematisch äquivalent mit GMRES für symmetrische Matrizen.

[2] es ist nicht trivial

Algorithmus 5.8 Conjugate Residual(CR)

Beschreibung: Input: A, b, x_0 m; Output x_m

bestimme $r_0 = b - Ax_0$, $p_0 = r_0$
for $j = 0, \ldots, m-1$ **do**
  $\alpha_j = (r_j, Ar_j)/(Ap_j, Ap_j)$
  $x_{j+1} = x_j + \alpha_j p_j$
  $r_{j+1} = r_j - \alpha_j Ap_j$
  $\beta_j = (r_{j+1}, Ar_{j+1})/(r_j, Ar_j)$
  $p_{j+1} = r_{j+1} + \beta_j p_j$
  $Ap_{j+1} = Ar_{j+1} + \beta_j Ap_j$ {Ap_j und Ar_{j+1} sind bereits berechnet!}
end for

5.1.8 BCG

Sei A unsymmetrisch. Seien $v_1, w_1 \in \mathbb{R}^n$, sodass $(v_1, w_1) = 1$. Sei

$$\mathcal{K}_m = \mathcal{K}_m(A, v_1) = span\{v_1, Av_1, A^2v_1, \ldots, A^{m-1}v_1\}$$

und

$$\mathcal{L}_m = \mathcal{K}_m(A, w_1) = span\{w_1, A^Tw_1, (A^T)^2w_1, \ldots, (A^T)^{m-1}w_1\}\,.$$

Der Lanczos Biorthogonalisierungprozess bildet zwei Basen, $v_1, v_2, \ldots, v_m$ für $\mathcal{K}_m$ und $w_1, w_2, \ldots, w_m$ für $\mathcal{L}_m$, sodass die Vektoren biorthogonal zueinander stehen, d.h.

$$(\forall 1 \le i, j \le m), \quad (v_i, w_j) = \delta_{i,j} = \begin{cases} 1 & \text{falls } i = j \\ 0 & \text{falls } i \neq j \end{cases}$$

Algorithmus 5.9 Lanczos Biorthogonalisation

Beschreibung: Input: A, m und v_1, w_1 mit $(v_1, w_1) = 1$; Output $(v_j)_{j=1:m+1}$, $(w_j)_{j=1:m+1}$, $(\alpha_j)_{j=1:m}$, $(\beta_j)_{j=1:m+1}$ und $(\delta_j)_{j=1:m+1}$

setze $\beta_1 = \delta_1 = 0$, $v_0 = w_0 = 0$
for $j = 1, \ldots, m$ **do**
  $\alpha_j = (Av_j, w_j)$
  $\widehat{v}_{j+1} = Av_j - \alpha_j v_j - \beta_j v_{j-1}$
  $\widehat{w}_{j+1} = A^Tw_j - \alpha_j w_j - \delta_j w_{j-1}$
  $\delta_{j+1} = |(\widehat{v}_{j+1}, \widehat{w}_{j+1})|^{1/2}$. Falls $\delta_{j+1} = 0$ Stop!
  $\beta_{j+1} = (\widehat{v}_{j+1}, \widehat{w}_{j+1})/\delta_{j+1}$
  $v_{j+1} = \widehat{v}_{j+1}/\delta_{j+1}$
  $w_{j+1} = \widehat{w}_{j+1}/\beta_{j+1}$
end for

Definiere (ähnlich wie bei Lanczos)

$$T_m = \begin{pmatrix} \alpha_1 & \beta_2 & & & \\ \delta_2 & \alpha_2 & \beta_3 & & \\ & \cdot & \cdot & \cdot & \\ & & \delta_{m-1} & \alpha_{m-1} & \beta_m \\ & & & \delta_m & \alpha_m \end{pmatrix}$$

Satz 5.7 ([58]) *Für die Ausgabe des obigen Algorithmus gilt:*

- $(\forall 1 \leq i, j \leq m+1), \quad (v_i, w_j) = \delta_{i,j}$
- $(v_j)_{j=1:m}$ *ist eine Basis für* $\mathcal{K}_m$, $(w_j)_{j=1:m}$ *ist eine Basis für* $\mathcal{L}_m$ *und*

$$\begin{aligned} AV_m &= V_m T_m + \delta_{m+1} v_{m+1} e_m^T, \\ A^T W_m &= W_m T_m^T + \beta_{m+1} w_{m+1} e_m^T, \\ W_m^T A V_m &= T_m \end{aligned} \tag{5.8}$$

mit $V_m = [v_1, \ldots, v_m]$ *und* $W_m = [w_1, \ldots, w_m]$

Erinnern wir uns an die Gleichungen (5.4) und (5.5), so erkennen wir aus (5.8), dass T_m den Operator bei der Petrov-Galerkin Projektion repräsentiert. Mit den Spaltenvektoren von V_m und W_m als Basen, haben wir eine einfache Projektionsmethode (vom Typ 5.1) mit $\mathcal{K} = \mathcal{K}_m$ und $\mathcal{L} = \mathcal{L}_m$.

Ähnlich wie beim CG-Verfahren, kann man[3] die oben entstandene iterative Methode (wie bei der allgemeine Projektionsmethode, Seite 87) mathematisch äquivalent umschreiben:

Algorithmus 5.10 Biconjugate Gradient(BCG)

Beschreibung: Input: A, b, x_0 m; Output x_m

1: bestimme $r_0 = b - Ax_0$. Wähle ein $r_0^* \in \mathbb{R}^n$ sodass $(r_0, r_0^*) \neq 0$
2: setze $p_0 = r_0$, $p_0^* = r_0^*$
3: **for** $j = 0, \ldots, m-1$ **do**
4: $\quad \alpha_j = (r_j, r_j^*)/(Ap_j, p_j^*)$
5: $\quad x_{j+1} = x_j + \alpha_j p_j$
6: $\quad r_{j+1} = r_j - \alpha_j A p_j$
7: $\quad r_{j+1}^* = r_j^* - \alpha_j A^T p_j^*$
8: $\quad \beta_j = (r_{j+1}, r_{j+1}^*)/(r_j, r_j^*)$
9: $\quad p_{j+1} = r_{j+1} + \beta_j p_j$
10: $\quad p_{j+1}^* = r_{j+1}^* + \beta_j p_j^*$
11: **end for**

[3] nicht trivial([58])

Dieser Algorithmus löst im Prinzip zwei LGS, $Ax = b$ und $A^T x^* = b^*$, wenn man noch die Zeile $x^*_{j+1} = x^*_j + \alpha_j p^*_j$ ergänzt und r^*_0, x^*_0 und b^* die Beziehung $r^*_0 = b^* - A^T x^*_0$ erfüllen.

BCG ist also eine Projektionsmethode mit $\mathcal{K} = \mathcal{K}_m$ und $\mathcal{L} = \mathcal{L}_m$.

5.1.9 Transponierungsfreie Methoden

Mit BCG als Ausgangspunkt sind zwei weitere iterative Methoden entwickelt worden, CGS ([63]) und BICGSTAB[4] ([66]). Beide machen **keinen** Gebrauch von A^T.

Beim BCG genügt das j-te Residuum r_j:

$$r_j = \phi_j(A) r_0$$

wobei $\phi_j(t)$ ein Polynom j-en Grades ist mit $\phi_j(0) = 1$.

CGS bildet eine iterative Lösungsfolge $x^{CGS}_0 = x_0, x^{CGS}_1, \ldots, x^{CGS}_j, \ldots$, sodass ihre zugehörige Residuumsfolge $r^{CGS}_0 = r_0, r^{CGS}_1, \ldots, r^{CGS}_j, \ldots$, der Gleichung

$$r^{CGS}_j = \phi^2_j(A) r_0$$

genügt.

BICGSTAB bildet eine iterative Lösungsfolge $x^{bicgstab}_0 = x_0, x^{bicgstab}_1, \ldots, x^{bicgstab}_j, \ldots$, sodass ihre zugehörige Residuumsfolge $r^{bicgstab}_0 = r_0, r^{bicgstab}_1, \ldots, r^{bicgstab}_j, \ldots$, der Gleichung

$$r^{bicgstab}_j = \psi_j(A)\phi_j(A) r_0$$

genügt. Das Polynom $\psi_j(t)$ besitzt Grad j, $\psi_j(0) = 1$ und $\psi_j(t) = (1 - \omega_j t)\psi_{j-1}(t)$. Die Konstanten ω_j werden so gewählt, dass eine bestimmte Glättungsbedingung erfüllt wird.

Für eine Herleitung beider Algorithmen und ihre Formulierung verweisen wir auf [58].

5.2 Präkonditionierung

Es gibt keine mathematisch klare Definition für die Präkonditionierung. Bezogen auf ein lineares Gleichungssystem $Ax = b$, bedeutet Präkonditionierung die Umwandlung in ein äquivalentes LGS $Cy = d$, sodass letzteres numerisch einfacher zu lösen ist. Als Löser betrachtet man hauptsächlich iterative Krylov-Unterraum Methoden.

Typische Präkonditionierungstechniken:

[4] Conjugate Gradient Squared und Biconjugate Gradient Stabilized

- **Split Preconditioners**: Finde M_1, M_2 mit $A \approx M_1 M_2$. Danach betrachtet man

$$\underbrace{\left(M_1^{-1} A M_2^{-1}\right)}_{C} \underbrace{(M_2 x)}_{y} = \underbrace{\left(M_1^{-1} b\right)}_{d} .$$

  Die Matrizen M_1 und M_2 sollten so sein, dass die Multiplikationen $M_1^{-1} \times vector$, $M_2^{-1} \times vector$ und $M_2 \times vector$ billig zu bestimmen sind. ILUs[5] gehören zu dieser Klasse.

- **Approximate Inverse Preconditioners**: Finde M mit $M \approx A$, sodass $M^{-1} \times vector$ billig zu bestimmen ist. Es gibt zwei Varianten:

  1. Linke Präkonditionierung $\underbrace{(M^{-1} A)}_{C} \underbrace{(x)}_{y} = \underbrace{(M^{-1} b)}_{d}$.
  2. Rechte Präkonditionierung $\underbrace{(A M^{-1})}_{C} \underbrace{(M x)}_{y} = \underbrace{(b)}_{d}$. Dabei sollte auch $M \times vector$ billig zu bestimmen sein. Rechte Präkonditionierung ist außerdem Residuum-Invariant, d.h. $r_y = d - Cy = b - Ax = r_x$.

- **Factorized Approximate Inverse Preconditioners**: Finde W, Z, D mit $W^T A Z \approx D$, D Diagonalmatrix. Wieder gibt es zwei Varianten:

  1. mit D: $\underbrace{(W^T A Z D)}_{C} \underbrace{(D^{-1} Z^{-1} x)}_{y} = \underbrace{(W^T b)}_{d}$.
  2. ohne D: $\underbrace{(W^T A Z)}_{C} \underbrace{(Z^{-1} x)}_{y} = \underbrace{(W^T b)}_{d}$.

  Dabei sollten $W^T \times vector$, $Z \times vector$ und $Z^{-1} \times vector$ billig zu bestimmen sein.

5.2.1 ILU(P)

Sei $P_n = \{(i,j) : (1 \leq i,j \leq n) \wedge (i \neq j)\}$ und $P \subseteq P_n$ das sog. Zero-Pattern. Folgender Satz von Meijerink und van der Vorst ([50],[58]) hat sehr zur Verbreitung der ILU-Methoden beigetragen.

Zur Erinnerung: Sei $B \in \mathbb{R}^{n \times n}$. Eine **Zerlegung** $B = T - Q$ heißt **regulär**, falls T nichtsingulär ist und $T^{-1}, Q \geq 0$ nichtnegative Matrizen sind. Solche Zerlegungen sind insbesondere sinnvoll für M-Matrizen. Ist B eine M-Matrix, so konvergiert die iterative Folge $x_{k+1} = T^{-1} Q x_k + T^{-1} c$ gegen die (eindeutige) Lösung des linearen Gleichungsystems $Bx = c$.

[5] Unvollständige LU Zerlegung

Satz 5.8 ([50]) *Falls A eine M-Matrix ist, dann gibt es eine untere Dreiecksmatrix mit Einheitsdiagonale L, eine obere Dreiecksmatrix U und eine Matrix R, sodass:*

$$A = LU - R$$

wobei

$$\forall (i,j) \in P,\ L_{i,j} = 0 \qquad (L\Big|_P = 0),$$
$$\forall (i,j) \in P,\ U_{i,j} = 0 \qquad (U\Big|_P = 0),$$
$$\forall (i,j) \notin P,\ R_{i,j} = 0 \qquad (R\Big|_{P^C} = 0).$$

Darüber hinaus sind die Faktoren L und U eindeutig und die Zerlegung $A = LU - R$ regulär.

Dass dieser Satz Garantien nur für den M-Matrix Fall gibt, hindert uns nicht daran, zu versuchen, solche Zerlegungen auch für beliebige Matrizen zu berechnen.

Algorithmus 5.11 General Static Pattern ILU — KIJ Variante

Beschreibung: Input: A, P; Output L, U "in place" sodass $L|_P = U|_P = 0$ und $(LU)|_{P^C} = A|_{P^C}$

```
1: for k = 1, ..., n − 1 do
2:   for i = k + 1, ..., n und (i, k) ∉ P do
3:     a_{i,k} = a_{i,k} / a_{k,k}
4:     for j = k + 1, ..., n und (k, j) ∉ P do
5:       if (i, j) ∉ P then
6:         a_{i,j} = a_{i,j} − a_{i,k} a_{k,j}
7:       end if
8:     end for
9:   end for
10: end for
```

Implementierungsfreundlicher ist aber die IKJ Variante (die Ausgabe ist sowohl arithmetisch als auch numerisch identisch):

Algorithmus 5.12 General Static Pattern ILU - IKJ Variante

Beschreibung: Input: A, P; Output L, U "in place" sodass $L|_P = U|_P = 0$ und $(LU)|_{P^C} = A|_{P^C}$

```
1: for i = 2, ..., n do
2:   for k = 1, ..., i − 1 und (i, k) ∉ P do
3:     a_{i,k} = a_{i,k} / a_{k,k}
4:     for j = k + 1, ..., n und (k, j) ∉ P do
5:       if (i, j) ∉ P then
6:         a_{i,j} = a_{i,j} − a_{i,k} a_{k,j}
7:       end if
8:     end for
9:   end for
10: end for
```

5.2.2 ILU(0)

Für $P^C = \text{Struct}(A)$ erhält man den sog. ILU(0) Präkonditionierer. Das Ergebnis ist eine unvollständige Zerlegung $A \approx LU$, sodass $\text{Struct}(L+U) = \text{Struct}(A)$ und $(\forall (i,j) \in \text{Struct}(A))((LU)_{i,j} = A_{i,j})$.

Algorithmus 5.13 ILU(0) - IKJ Variante

Beschreibung: Input: A; Output L, U "in place"

```
1: for i = 2, ..., n do
2:   for k = 1, ..., i − 1 und (i, k) ∈ Struct(A) do
3:     a_{i,k} = a_{i,k} / a_{k,k}
4:     for j = k + 1, ..., n und (k, j) ∈ Struct(A) do
5:       if (i, j) ∈ Struct(A) then
6:         a_{i,j} = a_{i,j} − a_{i,k} a_{k,j}
7:       end if
8:     end for
9:   end for
10: end for
```

5.2.3 ILU(p)

ILU(p) basiert auf Satz 4.2. Definiere die Menge P^C hier als:

$$P_p^C = \Big\{(i,j) : \text{ es gibt in } G(A) \text{ einen (gerichteten) Weg von } i, \text{ über } \{1,2,\ldots,\min(i,j)-1\}, \text{ nach } j \text{ mit höchstens } p \text{ Zwischenknoten}\Big\}$$

ILU(0) kann als ein Sonderfall von ILU(p) betrachtet werden, nämlich für $p = 0$.

5.2.4 ILUT

Unter einer Droppingregel verstehen wir eine Regel oder ein Kriterium, nach dem ein Eintrag während der Berechnung einer ILU auf Null gesetzt wird. Bei ILU(P) war diese Regel "Die Zugehörigkeit zu P". Der ILU(P) Präkonditionierer basiert ausschließlich auf einem vorgegebenen statischen Droppingpattern. Er nimmt keine Rücksicht auf numerische Werte. Ein guter Präkonditionierer sollte dies tun, damit nicht große Einträge auf Null gesetzt werden. ILUT ([57],[58]) ist ein Schritt in diese Richtung. Der folgende allgemeine ILUT-Algorithmus benutzt die IKJ Variante der Gauß-Elimination:

Algorithmus 5.14 ILUT

Beschreibung: Input: A; Output L, U

1: **for** $i = 1,\ldots,n$ **do**
2: $w = a_{i,:}$ $\{w = (w_1,\ldots,w_n)\}$
3: **for** $k = 1,\ldots,i-1$ **do**
4: **if** $w_k \neq 0$ **then**
5: $w_k = w_k/a_{k,k}$
6: wende eine Droppingregel auf w_k an
7: **if** $w_k \neq 0$ **then**
8: $w_{k+1:n} = w_{k+1:n} - w_k u_{k,k+1:n}$
9: **end if**
10: **end if**
11: **end for**
12: wende eine Droppingregel auf w an
13: setze $l_{i,1:i-1} = w_{1:i-1}$ und $u_{i,i:n} = w_{i:n}$
14: setze $w_{:} = 0$
15: **end for**

ILU(P) ist eine Sonderfall von ILUT, mit der Droppingregel "setze einen Eintrag (i,j) auf Null, wenn $(i,j) \in P$".
Die ILUT(ρ,τ) erhält man mit der Droppingregel:

1. benutze in Zeile 6 die Droppingregel "setze w_k auf Null, falls $|w_k| < \tau\|a_{i,:}\|_2$"

2. benutze in Zeile 12 die Droppingregel "setze zuerst alle die Elemente auf Null, die kleiner als $\tau\|a_{i,:}\|_2$ sind; behalte anschließend nur die p größten Einträge jeweils von L und U". Auf das Diagonalelement w_i wird allerdings nie Dropping angewandt.

Betrachten wir genauer die Zeilen 3 und 4. Es ist nicht notwendig jedes Element aus $\{1, 2, \ldots, i-1\}$ einzeln zu betrachten. Wir brauchen eigentlich das nächstkleinste k mit $w_k \neq 0$. Daher ist eine priority queue Datenstruktur nötig, z.B ein Heap oder ein binärer Suchbaum, um diesen Algorithmus effizient zu implementieren.

5.2.5 ILUTP

Beim ILUT Algorithmus ist es möglich, direkt nach der elften Zeile, Pivotisierung durchzuführen. Man kann w_i mit einen beliebigen Element $w_{\bar{i}}$ aus $w_{i+1:n}$ permutieren. Dabei darf nicht vergessen werden, die Vertauschung der Spalten i und $\bar{i}$ sowohl beim (bisher entstandenen) U-Teil $U_{1:i-1,:}$ als auch beim (übriggebliebenen) A-Teil $A_{i+1:n,:}$ durchzuführen.

5.2.6 ILUS

Der ILUS Präkonditionierer stützt sich auf die folgende Beobachtung:

Sei $A_k = L_k D_k U_k$ eine $k \times k$ Matrix und $A_{k+1} = \begin{pmatrix} A_k & v_k \\ w_k & \alpha_{k+1} \end{pmatrix}$ eine $(k+1) \times (k+1)$ Matrix. Dann gilt für A_{k+1}:

$$A_{k+1} = \begin{pmatrix} L_k & 0 \\ y_k & 1 \end{pmatrix} \begin{pmatrix} D_k & 0 \\ 0 & d_{k+1} \end{pmatrix} \begin{pmatrix} U_k & z_k \\ 0 & 1 \end{pmatrix}$$

mit $z_k = D_k^{-1} L_k^{-1} v_k$, $y_k^T = D_k^{-1} (U_k^T)^{-1} w_k^T$ und $d_{k+1} = \alpha_{k+1} - y_k D_k z_k$.

Sei $E_k = I - L_k$. Es ist $E_k^k = 0$ weil E_k eine untere Dreiecksmatrix mit verschwindender Diagonale ist. Dadurch gilt $L_k^{-1} = (I - E_k)^{-1} = I + E_k + E_k^2 + \ldots + E_k^{k-1}$. Der ILUS$(\epsilon, p)$ Präkonditionierer schneidet diese (Neumann) Summe nach $p+1$ Termen ab. Dadurch wird

$$z_k = D_k^{-1} L_k^{-1} v_k \approx D_k^{-1} (I + E_k + E_k^2 + \ldots + E_k^p) v_k \tag{5.9}$$

approximiert. Alle Multiplikationen $E_k \times \textit{vector}$ werden in *sparse-sparse mode* ausgeführt. Ähnlich wird zur Approximation von y_k auch mit U_k^T vorgegangen.

Algorithmus 5.15 ILUS(ϵ, p)

Beschreibung: Input: A; Output L, D und U

1: setze $A_1 = D_1 = a_{1,1}$ und $L_1 = U_1 = 1$
2: **for** $i = 1, \ldots, n-1$ **do**
3: berechne z_k nach (5.9)
4: berechne y_k entsprechend
5: wende eine Droppingregel auf z_k und y_k an
6: berechne $d_{k+1} = \alpha_{k+1} - y_k D_k z_k$
7: **end for**

5.2.7 Factorized Approximate Inverse

Sei $L_k A_k U_k = D_k$ eine $k \times k$ Matrix und $A_{k+1} = \begin{pmatrix} A_k & v_k \\ w_k & \alpha_{k+1} \end{pmatrix}$ eine $(k+1) \times (k+1)$ Matrix. Suche die nächste Zerlegung $L_{k+1} A_{k+1} U_{k+1} = D_{k+1}$:

$$\begin{pmatrix} L_k & 0 \\ -y_k & 1 \end{pmatrix} \begin{pmatrix} A_k & v_k \\ w_k & \alpha_{k+1} \end{pmatrix} \begin{pmatrix} U_k & -z_k \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} D_k & 0 \\ 0 & \delta_{k+1} \end{pmatrix}$$

Dann müssen y_k, z_k und δ_{k+1} folgende Gleichungen erfüllen:

$$A_k z_k = v_k \tag{5.10}$$
$$A_k^T y_k^T = w_k^T \tag{5.11}$$
$$\delta_{k+1} = \alpha_{k+1} - w_k z_k = \alpha_{k+1} - y_k w_k \tag{5.12}$$

Zur Lösung von (5.10) und (5.11) käme BCG in Frage, da wir es mit zwei LGS zu tun haben, deren Matrizen Transponierte von einander sind. Eine andere Möglichkeit ist, folgende Gleichungen zu benutzen:

$$z_k = (U_k D_k^{-1} L_k) v_k \,,$$
$$y_k^T = (L_k^T D_k^{-1} U_k^T) w_k^T \,.$$

Algorithmus 5.16 Factorized Approximate Inverse

Beschreibung: Input: A; Output L, D und U

1: **for** $i = 1, \ldots, n$ **do**
2: finde eine Näherungslösung für (5.10)
3: finde eine Näherungslösung für (5.11)
4: berechne $\delta_{k+1} = \alpha_{k+1} - w_k z_k - y_k v_k + y_k A_k z_k$ {neutraler als 5.12}
5: **end for**

5.2.8 AINV

AINV gehört zur "Factorized Approximate Inverse Preconditioners"-Klasse. Als Basis diente "the biconjugation algorithm". Der bildet zwei obere Dreiecksmatrizen, mit Einheitsdiagonale, $W = [w_1, w_2, \ldots, w_n]$ und $Z = [z_1, z_2, \ldots, z_n]$, sodass:

$$W^T AZ = D = \begin{pmatrix} p_1 & 0 & \ldots & 0 \\ 0 & p_2 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & p_n \end{pmatrix} \tag{5.13}$$

Es gilt $W^T AZ = (W^T A)Z = W^T(AZ)$. Falls $W^T A$ obere und AZ untere Dreiecksmatrizen sind, dann ist 5.13 erfüllt. Der "biconjugation" Algorithmus ([34]) benutzt diese Überlegung und erzeugt W, Z, sodass für alle $i = 1, 2, \ldots, n$:

- Die Zeile $(W^T)_{i,:}$, d.h. die Spalte $W_{:,i}$, senkrecht zu $A_{:,1}, A_{:,2}, \ldots, A_{:,i-1}$ steht
- Die Spalte $Z_{:,i}$, senkrecht zu $A_{1,:}, A_{2,:}, \ldots, A_{i-1,:}$ steht

Die Gleichung $W^T AZ = D$ kann äquivalent wie folgt geschrieben werden:

$$\forall i \neq j, \quad w_i^T A z_j = 0,$$

d.h. die Spalten von W und Z sind A-orthogonal.

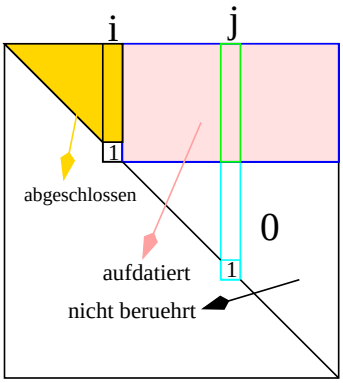
Algorithmus 5.17 The Biconjugation Algorithm

Beschreibung: Input: A; Output W, Z und D

1: $\forall i = 1, \ldots, n$, setze $w_i^{(0)} = z_i^{(0)} = e_i$, {d.h. initialisiere $W = Z = I_n$}
2: {jeder w_i bzw. z_i durchläuft $w_i^{(0)}, \ldots, w_i^{(i-1)}$ bzw. $z_i^{(0)}, \ldots, z_i^{(i-1)}$}
3: **for** $i = 1 : n$ **do**
4: **for** $j = i : n$ **do**
5: $p_j^{(i-1)} = A_{i,:} z_j^{(i-1)}$, $q_j^{(i-1)} = (A_{:,i})^T w_j^{(i-1)}$,
6: **end for**
7: **for** $j = i + 1 : n$ **do**
8: (1) $w_j^{(i)} = w_j^{(i-1)} - \left(q_j^{(i-1)} / q_i^{(i-1)}\right) w_i^{(i-1)}$ $\{\perp A_{:,1}, ..., A_{:,i}\}$
9: (2) $z_j^{(i)} = z_j^{(i-1)} - \left(p_j^{(i-1)} / p_i^{(i-1)}\right) z_i^{(i-1)}$ $\{\perp A_{1,:}, ..., A_{i,:}\}$
10: **end for**
11: **end for**
12: setze $W = \left[w_1^{(0)}, w_2^{(1)}, \ldots, w_n^{(n-1)}\right]$, $Z = \left[z_1^{(0)}, z_2^{(1)}, \ldots, z_n^{(n-1)}\right]$
13: setze $D = \operatorname{diag}\left(\left(p_1^{(0)}, p_2^{(1)}, \ldots, p_n^{(n-1)}\right)\right)$ $\{\equiv \operatorname{diag}\left(\left(q_1^{(0)}, q_2^{(1)}, \ldots, q_n^{(n-1)}\right)\right)\}$

Lemma 5.2 ([34]) *Für $i = 1, \ldots, n-1$ gilt am Ende der i-ten Schleife:*

$$\forall j > i, \quad w_j^{(i)} \perp A_{:,1}, A_{:,2}, \ldots, A_{:,i}$$
$$\forall j > i, \quad z_j^{(i)} \perp A_{1,:}, A_{2,:}, \ldots, A_{i,:}$$



Nach [15] und [14] können im obigen Algorithmus die Vektoren p und q durch eine etwas andere Formel bestimmt werden. Arithmetisch gesehen liefert die neue Variante identische Werte für p und q:

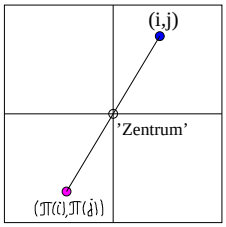
Algorithmus 5.18 The Biconjugation Algorithm, vektorisierte Schreibweise

Beschreibung: Input: A; Output W, Z und D

1: $p = q = zeros(1, n)$, $W = Z = I_n$
2: **for** $i = 1 : n$ **do** {Spaltendurchlauf}
3: $\quad p_{i:n} = ((W^T AZ)_{i:n,i})^T = (Z^T A^T W)_{i,i:n} = (Z_{:,i})^T A^T W_{:,i:n}$
4: $\quad q_{i:n} = (W^T AZ)_{i,i:n} = (W_{:,i})^T AZ_{:,i:n}$
5: $\quad p_{i+1:n} = p_{i+1:n}/p_i$, $q_{i+1:n} = q_{i+1:n}/q_i$, $D_{i,i} = p_i \equiv q_i$
6: $\quad W_{1:i,i+1:n} = W_{1:i,i+1:n} - W_{1:i,i} p_{i+1:n}$ {Rank-1 update}
7: $\quad Z_{1:i,i+1:n} = Z_{1:i,i+1:n} - Z_{1:i,i} q_{i+1:n}$ {Rank-1 update}
8: **end for**

5.2.8.1 Über die Struktur von W und Z

Satz 5.9 ([37]) *Falls $G(A)$ stark zusammenhängend ist und* diag(A) *keine Nullen enthält, dann ist A^{-1}* ***voll*** *(d.h.* Struct$(A^{-1}) = \{(i,j) : 1 \leq i, j \leq n\}$*), egal wie sparse A ist.*



Sei $\pi = (n, n-1, \ldots, 2, 1)$ und P_π die zugehörige Permutationsmatrix. Für jede $n \times n$ Matrix B, repräsentiert die permutierte Matrix $P_\pi B P_\pi^T$ eine Symmetrie bezüglich des "Matrixzentrums". Aus $W^T A Z = D$ folgt $A^{-1} = Z D^{-1} W^T \Rightarrow P_\pi A^{-1} P_\pi^T = (P_\pi Z P_\pi^T)(P_\pi D^{-1} P_\pi^T)(P_\pi W^T P_\pi^T)$. Die drei Klammern der rechten Seite stellen die LDU-Zerlegung von $P_\pi A^{-1} P_\pi^T$ dar. Ist $G(A)$ stark zusammenhängend und $\mathrm{diag}(A)$ nullfrei, dann ist nach Satz 5.9 $P_\pi A^{-1} P_\pi^T$ voll. Weil in der LDU-Zerlegung einer voller Matrix die Dreiecksmatrizen L,U ebenfalls voll sind, ist es zu erwarten, dass W und Z relativ volle (obere Dreiecks) Matrizen sind.

5.2.8.2 Führe dropping ein

Den ursprünglichen AINV Algorithmus von Benzi und Tuma ([9]) erhält man, indem man beim "Biconjugation Algorithm" (Algorithmus 5.17) eine Droppingregel anwendet. Die AINV-ILU Beziehungen, die im nächsten Kapitel vorgestellt werden, basieren auf der zweiten Variante (Algorithmus 5.18). Entsprechend wird dort eine Droppingregel angewandt. Das entstehende Verfahren (Algorithmus 5.19) ist insofern interessant, als dass man die W^T und Z annähernd als genaue Inverse von L und U betrachten kann, mit L und U als Resultat einer ILU. Siehe dazu die Sätze 6.2 und 6.3

Algorithmus 5.19 AINV, vektorisierte Schreibweise

Beschreibung: Input: A; Output W, Z und D

1: $p = q = zeros(1, n)$, $W = Z = I_n$
2: **for** $i = 1 : n$ **do** {Spaltendurchlauf}
3: $p_{i:n} = ((W^T A Z)_{i:n,i})^T = (Z^T A^T W)_{i,i:n} = (Z_{:,i})^T A^T W_{:,i:n}$
4: $q_{i:n} = (W^T A Z)_{i,i:n} = (W_{:,i})^T A Z_{:,i:n}$
5: $p_{i+1:n} = p_{i+1:n}/p_i$, $q_{i+1:n} = q_{i+1:n}/q_i$
6: wende eine Droppingregel auf $p_{i+1:n}$ und $q_{i+1:n}$ an
7: $W_{1:i,i+1:n} = W_{1:i,i+1:n} - W_{1:i,i} p_{i+1:n}$ {Rank-1 update}
8: $Z_{1:i,i+1:n} = Z_{1:i,i+1:n} - Z_{1:i,i} q_{i+1:n}$ {Rank-1 update}
9: wende eine Droppingregel auf $W_{1:i,i+1:n}$ und $Z_{1:i,i+1:n}$ an
10: **end for**

Auf Grund der Überlegungen aus Satz 5.9 ist zu erwarten, dass das Fill-in beim AINV viel höher ist als bei ILU's.

Kapitel 6

Beziehungen ILU–AINV und das inverse based dropping

6.1 Zwei Varianten zur Schur-Aufdatierung

Sei B eine $k \times k$ Matrix und $A = \begin{pmatrix} B & F \\ E & C \end{pmatrix}$ eine $n \times n$ Matrix. Sei:

$$A = \begin{pmatrix} B & F \\ E & C \end{pmatrix} \approx \begin{pmatrix} \tilde{L}_B & 0 \\ \tilde{L}_E & I \end{pmatrix} \begin{pmatrix} \tilde{D}_B & 0 \\ 0 & \tilde{S} \end{pmatrix} \begin{pmatrix} \tilde{U}_B & \tilde{U}_F \\ 0 & I \end{pmatrix} = \begin{pmatrix} \tilde{L}_B\tilde{D}_B\tilde{U}_B & \tilde{L}_B\tilde{D}_B\tilde{U}_F \\ \tilde{L}_E\tilde{D}_B\tilde{U}_B & \tilde{S} + \tilde{L}_E\tilde{D}_B\tilde{U}_F \end{pmatrix} \tag{6.1}$$

eine unvollständige Zerlegung, wobei $\tilde{L}_B$, $\tilde{U}_B^T$ untere Dreiecksmatrizen mit Einheitsdiagonale und $\tilde{D}_B$ Diagonalmatrix sei. Das approximierte Schur-Komplement $\tilde{S}$ kann auf verschiedenen Wegen bestimmt werden. Die folgenden Zwei sind von besonderer Bedeutung:

- **S**-Variante:
$$\tilde{S}_S = C - \tilde{L}_E\tilde{D}_B\tilde{U}_F \tag{6.2}$$
- **T**-Variante ([65]):
$$\tilde{S}_T = \begin{pmatrix} -\tilde{L}_E\tilde{L}_B^{-1} & I \end{pmatrix} A \begin{pmatrix} -\tilde{U}_B^{-1}\tilde{U}_F \\ I \end{pmatrix} \tag{6.3}$$

Während die S-Variante naheliegend ist, scheint die T-Variante verwirrend zu sein. Folgende äquivalente Umformung motiviert diese Variante:

$$\tilde{S}_T = \left(\begin{pmatrix} \tilde{L}_B & 0 \\ \tilde{L}_E & I \end{pmatrix}^{-1} A \begin{pmatrix} \tilde{U}_B & \tilde{U}_F \\ 0 & I \end{pmatrix}^{-1} \right)_{k+1:n,k+1:n} \tag{6.4}$$

Explizit sieht das so aus:

$$\tilde{S}_T = C - \tilde{L}_E \tilde{L}_B^{-1} F - E\tilde{U}_B^{-1}\tilde{U}_F + \tilde{L}_E \tilde{L}_B^{-1} B \tilde{U}_B^{-1}\tilde{U}_F \,. \tag{6.5}$$

Betrachten wir genauer die Lage für $k = 1$. Sei

$$A = \begin{pmatrix} A_{1,1} & f \\ e & C \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ g & I \end{pmatrix} \begin{pmatrix} \delta & 0 \\ 0 & S \end{pmatrix} \begin{pmatrix} 1 & h \\ 0 & I \end{pmatrix} = \begin{pmatrix} \delta & \delta h \\ \delta g & S + \delta g h \end{pmatrix} .$$

In einer unvollständigen LDU wird Dropping auf g und h angewandt. Dadurch erhalten wir die approximierten Vektoren $\tilde{g} \approx g$ und $\tilde{h} \approx h$, also:

$$A = \begin{pmatrix} A_{1,1} & f \\ e & C \end{pmatrix} \approx \begin{pmatrix} 1 & 0 \\ \tilde{g} & I \end{pmatrix} \begin{pmatrix} \delta & 0 \\ 0 & \tilde{S} \end{pmatrix} \begin{pmatrix} 1 & \tilde{h} \\ 0 & I \end{pmatrix} = \begin{pmatrix} \delta & \delta\tilde{h} \\ \delta\tilde{g} & \tilde{S} + \delta\tilde{g}\tilde{h} \end{pmatrix}$$

Die **S**-Variante berechnet

$$\tilde{S}_S = C - \delta\tilde{g}\tilde{h} \,. \tag{6.6}$$

Seien $\hat{g} = g - \tilde{g}$ und $\hat{h} = h - \tilde{h}$ die Unterschiede zwischen den exakten und den approximierten Vektoren. Die **T**-Variante berechnet

$$\tilde{S}_T = C - \delta\tilde{g}h - \delta g\tilde{h} + \delta\tilde{g}\tilde{h} = C - \delta(\tilde{g}\tilde{h} + \hat{g}\tilde{h} + \tilde{g}\hat{h}) = C - \delta(gh - \hat{g}\hat{h}) \,. \tag{6.7}$$

Folgende Abbildung begründet wieso die Variante S bzw. T auch als AND bzw. OR[1] Variante bezeichnet wird.

g \ h	$\tilde{h}$	$\hat{h}$
$\tilde{g}$	S T	T
$\hat{g}$	T	

Wie zu erwarten, ist die S-Variante einfach zu implementieren, aber der Fehler ist groß, während die T-Variante aufwändiger zu implementieren, aber der Fehler kleiner ist.

Es folgt jetzt eine allgemeine ILU, welche uns helfen wird, eine Beziehung zu AINV herzustellen.

[1] bezüglich "nicht gedropped"

Algorithmus 6.1 ILU mit "on the fly" Berechnung der L^{-T} und U^{-1}

Beschreibung: Input: A; Output L, U, W, Z und D

1: setze $L = U = W = Z = I_n$, $S = A$, $p = q = zeros(1, n)$, $D_{1,1} = A_{1,1}$
2: **for** $i = 1 : n - 1$ **do**
3: setze $p = \left[zeros(1, i-1), (S_{i:n,i}/S_{i,i})^T\right]$
4: setze $q = \left[zeros(1, i-1), S_{i,i:n}/S_{i,i}\right]$
5: wende eine Droppingregel auf $p_{i+1:n}$ und $q_{i+1:n}$ an
6: setze $L_{:,i} = p^T$, $U_{i,:} = q$
7: $W = W\,(I_n - e_i p)$, $Z = Z\,(I_n - e_i q)$ {Rank-1 Update}
8: wende eine Droppingregel auf $W_{1:i,i+1:n}$ und $Z_{1:i,i+1:n}$ an
9: datiere das Schur-Komplement $S_{i+1:n,i+1:n}$ auf durch die S- oder T-Variante
10: setze $D_{i+1,i+1} = S_{i+1,i+1}$
11: **end for**

6.2 Beziehungen ILU–AINV

Um ein LGS durch eine ILU zu präkonditionieren, geht man typischerweise in folgenden Schritten vor:

1. bestimme eine ILU-Zerlegung: $A = LU + R$
2. bilde das präkond. LGS: $Cy = d$, wobei $C = L^{-1}AU^{-1} = I + L^{-1}RU^{-1}$
3. löse $Cy = d$ iterativ

Hier wird deutlich, dass $L^{-1}RU^{-1}$ wichtiger ist als R für die Güte des Präkonditionierers.

- Seien M_1 und M_2 zwei quadratische Matrizen und $M = M_1 M_2$. Stört man $M_1 + \Delta'$, $M_2 + \Delta''$, mit $\left|(\Delta^*)_{i,j}\right| \leq \varepsilon$, so entsteht bei M ein (komponentenweiser) Fehler: $\left|\left(M - (M_1 + \Delta')\,(M_2 + \Delta'')\right)_{i,j}\right| \leq \alpha\varepsilon^2 + \beta\varepsilon$.
- Bei M^{-1} wird dieser Fehler jedoch eine rationale Funktion bezüglich ε sein.
- Daraus resultiert die Erwartung, dass das herkömliche Dropping bei ILU gefährlicher ist als Dropping bei AINV. Berichte über die AINV-Robustheit unterstützen dies.

Wenn in den folgenden Sätzen (aus [15],[15],[12],[11],[16]) die Rede vom ILU bzw. AINV Algorithmus sein wird, dann ist der Algorithmus 6.1 bzw. 5.19 gemeint.

Satz 6.1 *Geht man wie folgt vor:*

a) wende die gleiche Droppingregel auf p und q beim AINV und ILU an,

b) wende keine Droppingregel auf W und Z beim AINV und ILU an,

c) wende die OR Variante des Schur-Updates bei ILU an,

dann gilt $W_{AINV} \equiv W_{ILU}$, $Z_{AINV} \equiv Z_{ILU}$, $L^{-T} = W$ *und* $U^{-1} = Z$.

Satz 6.2 *Sei* $\varepsilon > 0$. *Geht man wie folgt vor:*

a) ILU-Droppingregel für W und Z: $|W_{i,j}|, |Z_{i,j}| \leq \varepsilon$,

b) ILU-Schur-Update: entweder die AND oder die OR Variante,

dann gilt für den ILU-Algorithmus:

$$(\forall\, i \leq j),\ \left|\left(I - WL^T\right)_{i,j}\right| \leq (j-i)\,\varepsilon,\ \left|\left(I - ZU\right)_{i,j}\right| \leq (j-i)\,\varepsilon.$$

Geht man zusätzlich noch folgendermaßen vor:

c) berechne Schur-Update (bei der ILU) wie folgt :
$S_{i+1:n,i+1:n} = \left(W^T AZ\right)_{i+1:n,i+1:n} = \left(W_{:,i+1:n}\right)^T AZ_{:,i+1:n}$,

d) wende die gleiche Droppingregel (Punkt a) auch auf W_{AINV} *und* Z_{AINV} *an,*

e) wende keine Droppingregel auf p und q beim AINV und ILU an,

dann bekommt man identische Ergebnisse:
$W_{AINV} \equiv W_{ILU}$ *und* $Z_{AINV} \equiv Z_{ILU}$.

Satz 6.3 (Grundlage für 'Inverse Based Dropping') *Sei* $\varepsilon > 0$. *Geht man wie folgt vor:*

a) verwende bei AINV und ILU die Droppingregel: $|W_{i,j}|, |Z_{i,j}| \leq \varepsilon$,

b) wende keine Droppingregel auf p und q bei AINV und ILU an, aber im ILU-Schritt i:
drop $L_{j,i}$ *falls* $|L_{j,i}| \cdot \|W_{:,i}\|_\infty \leq \varepsilon$,
drop $U_{i,j}$ *falls* $|U_{i,j}| \cdot \|Z_{:,i}\|_\infty \leq \varepsilon$,

c) berechne Schur-Update (bei der ILU) wie folgt:
$S_{i+1:n,i+1:n} = \left(W^T AZ\right)_{i+1:n,i+1:n} = \left(W_{:,i+1:n}\right)^T AZ_{:,i+1:n}$,

dann erhält man: $W_{AINV} \equiv W_{ILU}$ *und* $Z_{AINV} \equiv Z_{ILU}$*, und es gilt:*
$(\forall i \leq j),\ \left|\left(L^{-T} - W\right)_{i,j}\right| \leq (2\,(j-i)-1)\,\varepsilon,\ \left|\left(U^{-1} - Z\right)_{i,j}\right| \leq (2\,(j-i)-1)\,\varepsilon.$

Die Aussage von Satz 6.3 kann man wie folgt interpretieren:

- Bekanntlich ist AINV robust. Mit den Drop-Regeln aus Satz 6.3 haben wir eine ILU, deren Inversen annähernd ein AINV Produkt sind. Das legt ILU-Robustheit nahe.
- Ein anderer wichtiger Anstoß: Weil ILU weniger Speicher braucht, könnte das als eine **komprimierte** $AINV$ betrachtet werden.
- W und Z sollten nie explizit berechnet werden. Wir wollen einen Algorithmus, der die ILU-Einfachheit und AINV-Robustheit kombiniert.

6.2.1 Implementierungsprobleme

- Die Normen $\|W_{:,i}\|_\infty$ und $\|Z_{:,i}\|_\infty$ sind nicht direkt berechenbar, wenn W und Z nicht explizit gespeichert werden.
- Dasselbe gilt für das Schur-Update: $S_{i+1:n,i+1:n} = \left(W^T A Z\right)_{i+1:n,i+1:n} = \left(W_{:,i+1:n}\right)^T A Z_{:,i+1:n}$.

Lösung:

- Aus $\left(W_{:,i}\right)^T \approx \left(L^{-1}\right)_{i,:}$ und $Z_{:,i} \approx \left(U^{-1}\right)_{:,i}$ versucht man im ILU-Schritt i, die beide Normen $\left\|\left(L^{-1}\right)_{i,:}\right\|_\infty$ und $\left\|\left(U^{-1}\right)_{:,i}\right\|_\infty$ abzuschätzen (siehe Condition Estimator 6.2.3).
- Ersetze das aufwendige Schur-Komplement durch ein approximiertes Schur-Komplement. Satz 6.3 empfiehlt die aufwendigere OR Variante. Diese Variante ist von Natur aus die naheliegendste Update-Variante. Die AND Variante ist schneller implementierbar, aber ungenauer.

6.2.2 Eine andere Herangehensweise

Sei $A \approx \tilde{L}\tilde{D}\tilde{U}$ eine ILU Zerlegung. $\tilde{L}^{-1}$ und $\tilde{U}^{-1}$ sind sehr wichtig für die Güte des Präkonditionierers (siehe Anfang von Abschnitt 6.2). Daraus folgt die Dropping-Strategie:

- Dropping sollte so gestaltet werden, dass keine dramatischen Veränderungen bei der Inversen entstehen.

Angenomen $(k-1)$ ILU-Schritte sind durchgeführt

- $A \approx \tilde{L}\tilde{D}\tilde{U}$
- $A \approx \begin{pmatrix} \tilde{L}_{11} & 0 & 0 \\ \tilde{L}_{21} & 1 & 0 \\ \tilde{L}_{31} & 0 & I_{n-k} \end{pmatrix} \begin{pmatrix} \tilde{D}_{11} & 0 & 0 \\ 0 & \tilde{S}_{22} & \tilde{S}_{23} \\ 0 & \tilde{S}_{32} & \tilde{S}_{33} \end{pmatrix} \begin{pmatrix} \tilde{U}_{11} & \tilde{U}_{12} & \tilde{U}_{13} \\ 0 & 1 & 0 \\ 0 & 0 & I_{n-k} \end{pmatrix}$
- $\tilde{L}^{-1} = W = \begin{pmatrix} W_{11} & 0 & 0 \\ W_{21} & 1 & 0 \\ W_{31} & 0 & I_{n-k} \end{pmatrix}$, $\tilde{U}^{-1} = Z = \begin{pmatrix} Z_{11} & Z_{12} & Z_{13} \\ 0 & 1 & 0 \\ 0 & 0 & I_{n-k} \end{pmatrix}$
- Führe den nächsten (k-ten) Schritt aus. Spalte und Zeile k werden bei $\tilde{D}$ eliminiert:
- $A \approx \tilde{L} \begin{pmatrix} I_{k-1} & 0 & 0 \\ 0 & 1 & 0 \\ 0 & \tilde{L}_{31} & I_{n-k} \end{pmatrix} \begin{pmatrix} \tilde{D}_{11} & 0 & 0 \\ 0 & \tilde{S}_{22} & 0 \\ 0 & 0 & \tilde{T}_{33} \end{pmatrix} \begin{pmatrix} I_{k-1} & 0 & 0 \\ 0 & 1 & \tilde{U}_{23} \\ 0 & 0 & I_{n-k} \end{pmatrix} \tilde{U}$
- $A \approx \begin{pmatrix} \tilde{L}_{11} & 0 & 0 \\ \tilde{L}_{21} & 1 & 0 \\ \tilde{L}_{31} & \tilde{L}_{32} & I_{n-k} \end{pmatrix} \begin{pmatrix} \tilde{D}_{11} & 0 & 0 \\ 0 & \tilde{S}_{22} & 0 \\ 0 & 0 & \tilde{T}_{33} \end{pmatrix} \begin{pmatrix} \tilde{U}_{11} & \tilde{U}_{12} & \tilde{U}_{13} \\ 0 & 1 & \tilde{U}_{23} \\ 0 & 0 & I_{n-k} \end{pmatrix}$
  wobei
  $\tilde{L}_{32} \approx \tilde{S}_{32}/\tilde{S}_{22}$, $\tilde{U}_{23} \approx \tilde{S}_{23}/\tilde{S}_{22}$, $\tilde{T}_{33} \approx \tilde{S}_{33} - \tilde{S}_{32}\tilde{S}_{23}/\tilde{S}_{22}$,
- Wie wird sich W ändern?
- $W \to \begin{pmatrix} I_{k-1} & 0 & 0 \\ 0 & 1 & 0 \\ 0 & \tilde{L}_{32} & I_{n-k} \end{pmatrix}^{-1} \begin{pmatrix} W_{11} & 0 & 0 \\ W_{21} & 1 & 0 \\ W_{31} & 0 & I_{n-k} \end{pmatrix} = \begin{pmatrix} W_{11} & 0 & 0 \\ W_{21} & 1 & 0 \\ W_{31} - \tilde{L}_{32}W_{21} & -\tilde{L}_{32} & I_{n-k} \end{pmatrix}$
- Der Fehler gegenüber dem exakten Update von W ist
  $\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ Error(\tilde{L}_{32}) \cdot W_{21} & Error(\tilde{L}_{32}) & 0 \end{pmatrix} = \begin{pmatrix} 0_{(k-1)\times 1} \\ 0_{1\times 1} \\ Error(\tilde{L}_{32}) \end{pmatrix} \begin{pmatrix} W_{21} & 1 & 0_{1\times(n-k)} \end{pmatrix}$
- Die (Rank-1) Matrix: $Error(\tilde{L}_{32}) \cdot \begin{bmatrix} W_{21} & 1 \end{bmatrix}$ macht also den Fehler aus
- Der Fehler $Error(\tilde{L}_{32})$ entsteht ausschließlich durch Dropping. Daher:
- Drop $\tilde{l}_{j,k}$, falls: $\left|\tilde{l}_{j,k}\right| \cdot \left\| \begin{bmatrix} W_{21} & 1 \end{bmatrix} \right\| = \left|\tilde{l}_{j,k}\right| \cdot \left\| e_k^T \tilde{L}^{-1} \right\| \leq \varepsilon$

- Ähnliches kann man auch für $\tilde{U}$ begründen. Ergebnis:
  Drop $\tilde{u}_{k,j}$ falls: $|\tilde{u}_{k,j}| \cdot \left\| \begin{bmatrix} Z_{12} \\ 1 \end{bmatrix} \right\| = |\tilde{u}_{k,j}| \cdot \left\| \tilde{U}^{-1} e_k \right\| \leq \varepsilon$

Satz 6.3 schlägt die Verwendung der Norm $\|\cdot\|_\infty$ vor.

6.2.3 Implementierung des Condition Estimator

Im Schritt k werden alle $\tilde{l}_{j,k}$ verworfen mit $\left|\tilde{l}_{j,k}\right| \cdot \left\| e_k^T \tilde{L}^{-1} \right\|_\infty \leq \varepsilon$. Wir brauchen eine Abschätzung für $\left\| e_k^T \tilde{L}^{-1} \right\|_\infty$. Möglich ist dies für $\left\| e_k^T \tilde{L}^{-1} \right\|_1$. Ersetzen wir $\|\|_\infty$ durch $\|\|_1$, so wird höchstens genau soviel verworfen, weil $\|\|_\infty \leq \|\|_1$.
Angenommen nach $(k-1)$ Schritten ist

$$\tilde{L} = \begin{pmatrix} \tilde{L}_{11} & 0 & 0 \\ \tilde{L}_{21} & 1 & 0 \\ \tilde{L}_{31} & 0 & I_{n-k} \end{pmatrix}$$

und $\tilde{L}_{32} = \left(\tilde{l}_{k+1,k}, \ldots, \tilde{l}_{n,k}\right)^T$ kommt im Schritt k als nächstes hinzu. Eigentlich hat $\tilde{L}_{32}$ keinen Einfluss auf $e_k^T \tilde{L}^{-1}$. Nur $\tilde{L}_{11}$ und $\tilde{L}_{21}$ sind relevant dafür. Es gilt: $(\forall b \in \mathbb{R}^n)$

$$\left\| e_k^T \tilde{L}^{-1} \right\|_1 \|b\|_\infty \geq \left| e_k^T \tilde{L}^{-1} b \right| = \left| e_k^T \left(\tilde{L}^{-1} b \right) \right| = \left| \left(\tilde{L}^{-1} b \right)_k \right|.$$

Falls der Vektor b **nur** aus Komponenten ± 1 besteht, gilt also $\left\| e_k^T \tilde{L}^{-1} \right\|_1 \geq \left| \left(\tilde{L}^{-1} b \right)_k \right|$. Andereseits gibt es einen solchen Vektor b, für den Gleichheit gilt. Wir sollten also versuchen, $\left| \left(\tilde{L}^{-1} b \right)_k \right|$ zu maximieren.
Die Idee ([20],[41]): b soll schrittweise um eine Komponente erweitert werden. Falls man $b_{1:k-1}$ kennt, dann wird b_k berechnet. Parallel dazu wird die k-te Komponente von $\tilde{L}^{-1} b$ berechnet. Diese wird als Schätzung für $\left\| e_k^T \tilde{L}^{-1} \right\|_1$ dienen.

$\tilde{L}^{-1} b$ ist die Lösung eines Systems mit unterer Dreiecksmatrix (und Einheitdiagonale). Man löst es durch Vorwärtssubstitution. Sei

$$\begin{pmatrix} \tilde{L}_{11} & 0 & 0 \\ \tilde{L}_{21} & 1 & 0 \\ \tilde{L}_{31} & \tilde{L}_{32} & I_{n-k} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix},$$

wobei $x_1, b_1 \in \mathbb{R}^{k-1}$ bekannt, $x_2, b_2 \in \mathbb{R}$ zu bestimmen und $x_3, b_3 \in \mathbb{R}^{n-k}$ für den Augenblick uninteressant sind (nicht zu vergessen, $b = (.., \pm 1, ..)^T$).

Es ist $x_2 = b_2 - \tilde{L}_{21}x_1$. Es gibt zwei Möglichkeiten: $b_2 = -1$ und $b_2 = +1$. Daraus ergeben sich zwei Werte für x_2, nämlich: x_2^- und x_2^+. Die einfachste Auswahl wäre das b_2, welches $|x_2|$ maximiert. Das wäre eine *Lokale* Strategie.
Global sollten wir einen look-ahead Blick auf $\left[\tilde{L}_{31}, \tilde{L}_{32}\right]$ werfen. Es ist

$$x_3 = b_3 - (\tilde{L}_{31}x_1 + \tilde{L}_{32}x_2).$$

Intuitiv ist klar, dass $\tilde{L}_{31}x_1 + \tilde{L}_{32}x_2$ wichtiger ist als x_2.

Um x_2 zu berechnen, wählen wir b_2 deshalb und damit x_2 so, dass $\left\|\tilde{L}_{31}x_1 + \tilde{L}_{32}x_2\right\|_1$ maximiert wird, aber wir wählen (lokal) $\max\left(|x_2^-|, |x_2^+|\right)$ als Konditionsschätzer für $\left\|e_k^T\tilde{L}^{-1}\right\|_1$.

Kostenmäßig ist die Berechnung von $\tilde{L}_{31}x_1 + \tilde{L}_{32}x_2^{\pm}$ problematisch. Aber, wenn wir $v = \tilde{L}_{31}x_1$ hätten, dann bräuchte v nur in den Komponenten aus $\mathrm{Struct}\left(\tilde{L}_{32}\right)$ aufdatiert zu werden. Weil $\tilde{L}_{32}$ sparse ist, ist dies billig. Um
$\max\left(\left\|\tilde{L}_{31}x_1 + \tilde{L}_{32}x_2^-\right\|_1, \left\|\tilde{L}_{31}x_1 + \tilde{L}_{32}x_2^+\right\|_1\right) = \max\left(\left\|v + \tilde{L}_{32}x_2^-\right\|_1, \left\|v + \tilde{L}_{32}x_2^+\right\|_1\right)$
zu bestimmen, braucht man nur die Indizes aus $\mathrm{Struct}\left(\tilde{L}_{32}\right)$ zu untersuchen. Tatsächlich kann man in einer Implementierung ganz auf x_1 verzichten, wenn man v entsprechend aufdatiert.

Alle diese Überlegungen münden in folgendem Algorithmus[12]:

Algorithmus 6.2 $\|\cdot\|_1$ cond. est. für die Zeilen der L^{-1}

1: $v = (0, 0, \ldots, 0)^T$, $CondEst = (1, 1, \ldots, 1)^T$,
2: **for** $k = 1 : n$ **do**
3: {lokal}
4: setze $x_+ = +1 - v_k$, $x_- = -1 - v_k$
5: $CondEst_k = \max\left(|x_-|, |x_+|\right)$ {es folgt: $CondEst_k \geq 1$}
6: {global}
7: setze $Q = \mathrm{Struct}\left(\tilde{l}_{k+1,k}, \ldots, \tilde{l}_{n,k}\right)$
8: setze $\nu_+ = \left\|v_Q + x_+\tilde{l}_{Q,k}\right\|_1$, $\nu_- = \left\|v_Q + x_-\tilde{l}_{Q,k}\right\|_1$
9: **if** $\nu_+ > \nu_-$ **then**
10: $v_Q = v_Q + x_+\tilde{l}_{Q,k}$
11: **else**
12: $v_Q = v_Q + x_-\tilde{l}_{Q,k}$
13: **end if**
14: **end for**

Kapitel 7

Eine neue ILU-Zerlegung mit inverse based dropping

In diesem Abschnitt zeige ich, wie man die multifrontale Methode und das inverse based dropping zu einer ILU mit Pivotisierung zusammenfügen kann. Es folgt zunächst ein ausführliches Beispiel von einer LDU-Zerlegung (mit Pivotisierung) durch die multifrontale Methode.

Um auf eine Matrix A die multifrontale Methode anwenden zu können, muss der Digraph $G(A)$ in einen zusammenhängenden Graph eingebettet sein. Selbstverständlich sollte dieser zusammenhängende Graph nicht beliebig sein, sondern so "klein" wie möglich, damit der Eliminationsbaum nicht ausartet (z.B. in eine Kette). Der kleinste Graph, der $G(A)$ enthält, ist $G(A + A^T)$. Er muss nicht zusammenhängend sein. Wenn $G(A)$ stark zusammenhängend ist, ist $G(A + A^T)$ jedoch immer zusammenhängend.

Der sicherste Weg, zu einen zusamenhängenden Graphen zu gelangen, ist die B-reduzible Normalform (Kapitel 2). Die Diagonalblöcke sind dann alle stark zusammenhängend. Daher habe ich mich in meinem Programm auf stark zusammenhängende Matrizen beschränkt.

Bei den Beispielmatrizen aus verschiedenen Sammlungen ergab sich die folgende Beobachtung: Es ist (fast) immmer so, dass die B-reduzible Normalform einen **dominanten** Diagonalblock besitzt, d.h. seine Dimension ist von der Größenordnung der Dimension der Anfangsmatrix. War die Anfangsmatrix nicht stark zusammenhängend, so habe ich immer das LGS betrachtet, welches diesen dominanten Diagonalblock als Matrix hat.

7.1 Ein multifrontales Beispiel

Als Beispiel habe ich die beidseitig irreduzible I-Matrix

$$A_e = \begin{pmatrix} 1 & & 0.8 & & 0.2 & & & & & \\ & 1 & 0.9 & & & & & & & 0.2 \\ 0.7 & & 1 & & 0.9 & & & & & 0.3 \\ & & & 1 & & & & & & -0.7 \\ -0.5 & & & -0.9 & 1 & & & & & -0.9 \\ & & & & & 1 & -0.3 & & 0.7 & \\ & & & & & 0.1 & 1 & & 0.1 & \\ & & & & & & & 1 & -0.8 & 0.6 \\ & & & & & -0.7 & 1 & & 1 & 0.5 \\ & 0.3 & -0.1 & -0.5 & 0.7 & & & 0.7 & & 1 \end{pmatrix}$$

ausgewählt[1]. Sie ist absichtlich relativ groß, damit man einen aussagekräftigen, nicht trivialen Eliminationsbaum erhält und dadurch die Methode besser beschrieben werden kann.

Es wird eine stabile LDU Zerlegung mit Pivotisierung gesucht:

$$P_\pi A_e P_\psi^T = LDU.$$

Da A_e noch nicht mal strukturell symmetrisch ist, wird sie in $\mathrm{Struct}\left(A_e + A_e^T\right)$ eingebettet:

$$A_e = \begin{pmatrix} 1 & & 0.8 & & 0.2 & & & & & \\ & 1 & 0.9 & & & & & & & 0.2 \\ 0.7 & 0.0 & 1 & & 0.9 & & & & & 0.3 \\ & & & 1 & 0.0 & & & & & -0.7 \\ -0.5 & & 0.0 & -0.9 & 1 & & & & & -0.9 \\ & & & & & 1 & -0.3 & & 0.7 & \\ & & & & & 0.1 & 1 & & 0.1 & \\ & & & & & & & 1 & -0.8 & 0.6 \\ & & & & & -0.7 & 1 & 0.0 & 1 & 0.5 \\ & 0.3 & -0.1 & -0.5 & 0.7 & & & 0.7 & 0.0 & 1 \end{pmatrix}.$$

Unten sind der (zusammenhängende) Graph $G_{A_e+A_e^T}$ und der Eliminationsbaum $T[A_e]$ abgebildet:

[1] I-Matrix, weil mein Programm die Matrizen zuerst in I-Matrizen umwandelt.

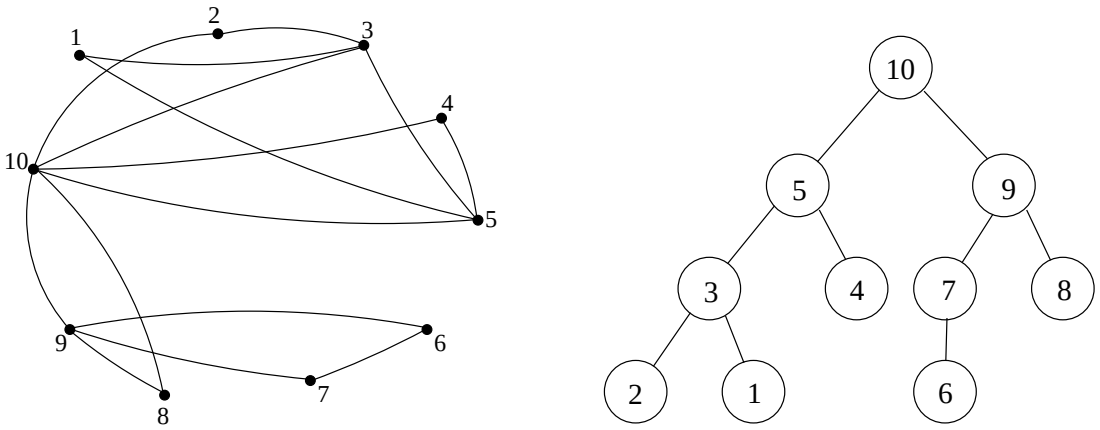


Abbildung 7.1: Der Graph $G_{A_e+A_e^T}$ und der Baum $T[A_e]$

Als Basis dient die multifrontale Methode (Seite 83). Außerdem wird hier delayed pivoting angewandt. Ich werde bei diesem Beispiel keine explizite Pivotisierungsstrategie anwenden. In erster Linie geht es mir darum zu verdeutlichen, wie die Methode funktioniert. Fangen wir an:

- Knoten 1: Er ist ein Blatt (keine Update-Matrizen aus früheren Schritten)

$$\mathcal{L}_1^* = \{1,3,5\},\ \mathcal{L}_1 = \{3,5\}$$

$$F_1 = \begin{pmatrix} A_{1,1} & A_{1,\mathcal{L}_1} \\ A_{\mathcal{L}_1,1} & 0 \end{pmatrix} = \begin{array}{c} \\ 1 \\ 3 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 3 & 5 \end{array} \\ \begin{pmatrix} 1 & 0.8 & 0.2 \\ 0.7 & 0 & 0 \\ -0.5 & 0 & 0 \end{pmatrix} \end{array}$$

  Der Knoten 1 ist totally summed und ist bereit für die Elimination. Das Pivot $Piv = 1$ sei z.B. akzeptabel. Durch dessen Elimination aus F_1 gewinnt man die Spalte(1) von L: $\begin{array}{c} 1 \\ 3 \\ 5 \end{array} \begin{pmatrix} 1 \\ 0.7 \\ -0.5 \end{pmatrix}$, die Zeile(1) von U: $\begin{array}{c} \begin{array}{ccc} 1 & 3 & 5 \end{array} \\ \begin{pmatrix} 1 & 0.8 & 0.2 \end{pmatrix} \end{array}$,
  die Update-Matrix: $U_1 = \begin{array}{c} \\ 3 \\ 5 \end{array} \begin{array}{c} \begin{array}{cc} 3 & 5 \end{array} \\ \begin{pmatrix} -0.56 & -0.14 \\ 0.4 & 0.1 \end{pmatrix} \end{array}$ mit $\mathcal{L}_1' = \{3,5\}$[2] und zuletzt $\pi(1) = 1$, $\psi(1) = 1$, $D(1) = 1$.

- Knoten 2: Er ist ein Blatt (keine Update-Matrizen aus früheren Schritten).

$$\mathcal{L}_2^* = \{2,3,10\},\ \mathcal{L}_2 = \{3,10\}$$

$$F_2 = \begin{pmatrix} A_{2,2} & A_{2,\mathcal{L}_2} \\ A_{\mathcal{L}_2,2} & 0 \end{pmatrix} = \begin{array}{c} \\ 2 \\ 3 \\ 10 \end{array} \begin{array}{c} \begin{array}{ccc} 2 & 3 & 10 \end{array} \\ \begin{pmatrix} 1 & 0.9 & 0.2 \\ 0.0 & 0 & 0 \\ 0.3 & 0 & 0 \end{pmatrix} \end{array}$$

[2] $\mathcal{L}_j' = \mathcal{L}_j^* - \{j\}$, falls j eliminiert wird und $\mathcal{L}_j' = \mathcal{L}_j^*$, falls j nicht eliminiert wird.

Der Knoten 2 ist totally summed und ist bereit für die Elimination. Das Pivot $Piv = 1$ sei z.B. akzeptabel. Durch dessen Elimination aus F_2 gewinnt man die Spalte(2) von L: $\begin{matrix}2\\3\\10\end{matrix}\begin{pmatrix}1\\0.0\\0.3\end{pmatrix}$, die Zeile(2) von U: $\begin{matrix}2&3&10\end{matrix}\begin{pmatrix}1&0.9&0.2\end{pmatrix}$, die Update-Matrix: $U_2 = \begin{matrix}&3&10\\3\\10\end{matrix}\begin{pmatrix}0.0&0.0\\-0.27&-0.06\end{pmatrix}$ mit $\mathcal{L}_2' = \{3, 10\}$, und zuletzt $\pi(2) = 2$, $\psi(2) = 2$, $D(2) = 1$.

- Knoten 3: Er besitzt die Knoten 1 und 2 als Söhne.

$$\mathcal{L}_3^* = \mathcal{L}_1' \cup \mathcal{L}_2' \cup \{3, 5, 10\} = \{3, 5, 10\},\ \mathcal{L}_3 = \{5, 10\}$$

$$F_3 = \begin{pmatrix}A_{3,3} & A_{3,\mathcal{L}_3}\\ A_{\mathcal{L}_3,3} & 0\end{pmatrix} \oplus U_1 \oplus U_2 = \begin{matrix}&3&5&10\\3\\5\\10\end{matrix}\begin{pmatrix}0.44&0.76&0.3\\0.4&0.1&0.0\\-0.37&0.0&-0.06\end{pmatrix}$$

Der Knoten 3 ist totally summed und ist bereit für die Elimination. Das Pivot $Piv = 0.44$ sei z.B. akzeptabel. Durch dessen Elimination aus F_3 gewinnt man die Spalte(3) von L: $\begin{matrix}3\\5\\10\end{matrix}\begin{pmatrix}1\\0.9091\\-0.8409\end{pmatrix}$, die Zeile(3) von U: $\begin{matrix}3&5&10\end{matrix}\begin{pmatrix}1&1.7273&0.6818\end{pmatrix}$, die Update-Matrix: $U_3 = \begin{matrix}&5&10\\5\\10\end{matrix}\begin{pmatrix}-0.5909&-0.2727\\0.6391&0.1923\end{pmatrix}$ mit $\mathcal{L}_3' = \{5, 10\}$ und zuletzt $\pi(3) = 3$, $\psi(3) = 3$, $D(3) = 0.44$.

- Knoten 4: Er ist ein Blatt (keine Update-Matrizen aus früheren Schritten).

$$\mathcal{L}_4^* = \{4, 5, 10\},\ \mathcal{L}_4 = \{5, 10\}$$

$$F_4 = \begin{pmatrix}A_{4,4} & A_{4,\mathcal{L}_4}\\ A_{\mathcal{L}_4,4} & 0\end{pmatrix} = \begin{matrix}&4&5&10\\4\\5\\10\end{matrix}\begin{pmatrix}1&0.0&-0.7\\-0.9&0&0\\-0.5&0&0\end{pmatrix}$$

Der Knoten 4 ist totally summed und ist bereit für die Elimination. Das Pivot $Piv = 1$ sei z.B. akzeptabel. Durch dessen Elimination aus F_4 gewinnt man die Spalte(4) von L: $\begin{matrix}4\\5\\10\end{matrix}\begin{pmatrix}1\\-0.9\\-0.5\end{pmatrix}$, die Zeile(4) von U: $\begin{matrix}4&5&10\end{matrix}\begin{pmatrix}1&0.0&-0.7\end{pmatrix}$, die Update-Matrix: $U_4 = \begin{matrix}&5&10\\5\\10\end{matrix}\begin{pmatrix}0.0&-0.63\\0.0&-0.35\end{pmatrix}$ mit $\mathcal{L}_4' = \{5, 10\}$ und zuletzt $\pi(4) = 4$, $\psi(4) = 4$, $D(4) = 1$.

- Knoten 5: Er besitzt die Knoten 3 und 4 als Söhne.

$$\mathcal{L}_5^* = \mathcal{L}_3' \cup \mathcal{L}_4' \cup \{5,10\} = \{5,10\},\ \mathcal{L}_5 = \{10\}$$

$$F_5 = \begin{pmatrix} A_{5,5} & A_{5,\mathcal{L}_5} \\ A_{\mathcal{L}_5,5} & 0 \end{pmatrix} \oplus U_3 \oplus U_4 = \begin{array}{c} \\ 5 \\ 10 \end{array} \begin{array}{c} \begin{array}{cc} 5 & 10 \end{array} \\ \begin{pmatrix} 0.4091 & -1.8027 \\ 1.3391 & -0.1577 \end{pmatrix} \end{array}$$

Der Knoten 5 ist totally summed und ist bereit für die Elimination. Das Pivot $Piv = 0.4091$ sei z.B. **nicht** akzeptabel. Dessen Elimination wird auf einen späteren Zeitpunkt verschoben (verspätet, verzögert).

$$U_5 = F_5 = \begin{array}{c} \\ 5 \\ 10 \end{array} \begin{array}{c} \begin{array}{cc} 5 & 10 \end{array} \\ \begin{pmatrix} 0.4091 & -1.8027 \\ 1.3391 & -0.1577 \end{pmatrix} \end{array}$$

mit $\mathcal{L}_5' = \{5,10\}$. Verspätete Knoten werden rot gefärbt.

- Knoten 6: Er ist ein Blatt (keine Update-Matrizen aus früheren Schritten).

$$\mathcal{L}_6^* = \{6,7,9\},\ \mathcal{L}_6 = \{7,9\}$$

$$F_6 = \begin{pmatrix} A_{6,6} & A_{6,\mathcal{L}_6} \\ A_{\mathcal{L}_6,6} & 0 \end{pmatrix} = \begin{array}{c} \\ 6 \\ 7 \\ 9 \end{array} \begin{array}{c} \begin{array}{ccc} 6 & 7 & 9 \end{array} \\ \begin{pmatrix} 1 & -0.3 & 0.7 \\ 0.1 & 0 & 0 \\ -0.7 & 0 & 0 \end{pmatrix} \end{array}$$

Der Knoten 6 ist totally summed und ist bereit für die Elimination. Das Pivot $Piv = 1$ sei z.B. akzeptabel. Durch dessen Elimination aus F_6 gewinnt man die Spalte(5)[3] von L: $\begin{array}{c} 6 \\ 7 \\ 9 \end{array} \begin{pmatrix} 1 \\ 0.1 \\ -0.7 \end{pmatrix}$, die Zeile(5) von U: $\begin{array}{c} \begin{array}{ccc} 6 & 7 & 9 \end{array} \\ \begin{pmatrix} 1 & -0.3 & 0.7 \end{pmatrix} \end{array}$,

die Update-Matrix: $U_6 = \begin{array}{c} \\ 7 \\ 9 \end{array} \begin{array}{c} \begin{array}{cc} 7 & 9 \end{array} \\ \begin{pmatrix} 0.03 & -0.07 \\ -0.21 & 0.49 \end{pmatrix} \end{array}$ mit $\mathcal{L}_6' = \{7,9\}$ und zuletzt $\pi(5) = 6$, $\psi(5) = 6$, $D(5) = 1$.

- Knoten 7: Er besitzt den Knoten 6 als Sohn.

$$\mathcal{L}_7^* = \mathcal{L}_6' \cup \{7,9\} = \{7,9\},\ \mathcal{L}_7 = \{9\}$$

$$F_7 = \begin{pmatrix} A_{7,7} & A_{7,\mathcal{L}_7} \\ A_{\mathcal{L}_7,7} & 0 \end{pmatrix} \oplus U_6 = \begin{array}{c} \\ 7 \\ 9 \end{array} \begin{array}{c} \begin{array}{cc} 7 & 9 \end{array} \\ \begin{pmatrix} 1.03 & 0.03 \\ 0.79 & 0.49 \end{pmatrix} \end{array}$$

Der Knoten 7 ist totally summed und ist bereit für die Elimination. Das Pivot $Piv = 1.03$ sei z.B. **nicht** akzeptabel. Dessen Elimination wird auf eine späteren Zeitpunkt verschoben (verspätet).

$$U_7 = F_7 = \begin{array}{c} \\ 7 \\ 9 \end{array} \begin{array}{c} \begin{array}{cc} 7 & 9 \end{array} \\ \begin{pmatrix} 1.03 & 0.03 \\ 0.79 & 0.49 \end{pmatrix} \end{array}$$

mit $\mathcal{L}_7' = \{7,9\}$.

[3] Vorsicht, nicht 6

- Knoten 8: Er ist ein Blatt (keine Update-Matrizen aus früheren Schritten).
$$\mathcal{L}_8^* = \{8,9,10\},\ \mathcal{L}_8 = \{9,10\}$$
$$F_8 = \begin{pmatrix} A_{8,8} & A_{8,\mathcal{L}_8} \\ A_{\mathcal{L}_8,8} & 0 \end{pmatrix} = \begin{array}{c} \\ 8 \\ 9 \\ 10 \end{array} \begin{array}{c} \begin{array}{ccc} 8 & 9 & 10 \end{array} \\ \begin{pmatrix} 1 & -0.8 & 0.6 \\ 0.0 & 0 & 0 \\ 0.7 & 0 & 0 \end{pmatrix} \end{array}$$
Der Knoten 8 ist totally summed und ist bereit für die Elimination. Das Pivot $Piv = 1$ sei z.B. akzeptabel. Durch dessen Elimination aus F_8 gewinnt man die Spalte(6)[4] von L: $\begin{array}{c} 8 \\ 9 \\ 10 \end{array} \begin{pmatrix} 1 \\ 0.0 \\ 0.7 \end{pmatrix}$, die Zeile(6) von U: $\begin{array}{c} \begin{array}{ccc} 8 & 9 & 10 \end{array} \\ \begin{pmatrix} 1 & -0.8 & 0.6 \end{pmatrix} \end{array}$,
die Update-Matrix: $U_8 = \begin{array}{c} \\ 9 \\ 10 \end{array} \begin{array}{c} \begin{array}{cc} 9 & 10 \end{array} \\ \begin{pmatrix} 0.0 & 0.0 \\ 0.56 & -0.42 \end{pmatrix} \end{array}$ mit $\mathcal{L}_8' = \{9,10\}$ und zuletzt $\pi(6) = 8$, $\psi(6) = 8$, $D(6) = 1$.

- Knoten 9: Er besitzt die Knoten 7 und 8 als Söhne.
$$\mathcal{L}_9^* = \mathcal{L}_7' \cup \mathcal{L}_8' \cup \{9,10\} = \{7,9,10\},\ \mathcal{L}_9 = \{7,10\}$$
$$F_9 = \begin{pmatrix} A_{9,9} & A_{9,\mathcal{L}_9} \\ A_{\mathcal{L}_9,9} & 0 \end{pmatrix} \oplus U_7 \oplus U_8 = \begin{array}{c} \\ 7 \\ 9 \\ 10 \end{array} \begin{array}{c} \begin{array}{ccc} 7 & 9 & 10 \end{array} \\ \begin{pmatrix} 1.03 & 0.03 & 0.0 \\ 0.79 & 1.49 & 0.5 \\ 0.0 & 0.56 & -0.42 \end{pmatrix} \end{array}$$
Die Knoten 7 und 9 sind totally summed und sind bereit für die Elimination. Da es beim Knoten 7 Pivot nichts geändert hat (ist immer noch 1.03), machen wir mit Knoten 9 weiter. Das Pivot $Piv = 1.49$ sei z.B. akzeptabel. Durch dessen Elimination aus F_9 gewinnt man die Spalte(7) von L: $\begin{array}{c} 7 \\ 9 \\ 10 \end{array} \begin{pmatrix} 0.0201 \\ 1 \\ 0.3758 \end{pmatrix}$, die Zeile(7) von U: $\begin{array}{c} \begin{array}{ccc} 7 & 9 & 10 \end{array} \\ \begin{pmatrix} 0.5320 & 1 & 0.3356 \end{pmatrix} \end{array}$, die Update-Matrix[5]: $U_9 = \begin{array}{c} \\ 7 \\ 10 \end{array} \begin{array}{c} \begin{array}{cc} 7 & 10 \end{array} \\ \begin{pmatrix} 1.0141 & -0.0101 \\ -0.2969 & -0.6079 \end{pmatrix} \end{array}$ mit $\mathcal{L}_9' = \{7,10\}$ und zuletzt $\pi(7) = 9$, $\psi(7) = 9$, $D(7) = 1.49$.

  Wie bereits erwähnt, ist der Knoten 7 totally summed. Ist das Pivot $Piv = 1.0141$ z.B. akzeptabel, kann es hier direkt (aus U_9) eliminiert werden. Die Spalte(8) von L: $\begin{array}{c} 7 \\ 10 \end{array} \begin{pmatrix} 1 \\ -0.2928 \end{pmatrix}$, die Zeile(8) von U: $\begin{array}{c} \begin{array}{cc} 7 & 10 \end{array} \\ \begin{pmatrix} 1 & -0.0099 \end{pmatrix} \end{array}$,
die Update-Matrix: $U_9 = \begin{array}{c} \\ 10 \end{array} \begin{array}{c} \begin{array}{c} 10 \end{array} \\ \begin{pmatrix} -0.6109 \end{pmatrix} \end{array}$ mit $\mathcal{L}_9' = \{10\}$ und zuletzt

[4] Vorsicht, nicht 7 oder 8

[5] Hier wurde die zweite Zeile und Spalte von F_9 eliminiert.

$\pi(8) = 7,\ \psi(8) = 7,\ D(8) = 1.0141.$

- Knoten 10: Er besitzt die Knoten 5 und 9 als Söhne.

$$\mathcal{L}^*_{10} = \mathcal{L}'_5 \cup \mathcal{L}'_9 \cup \{10\} = \{5, 10\},\ \mathcal{L}_{10} = \{5\}$$

$$F_{10} = \begin{pmatrix} A_{10,10} & A_{10,\mathcal{L}_{10}} \\ A_{\mathcal{L}_{10},10} & 0 \end{pmatrix} \oplus U_5 \oplus U_9 = \begin{matrix} \\ 5 \\ 10 \end{matrix} \begin{matrix} \begin{matrix} 5 & 10 \end{matrix} \\ \begin{pmatrix} 0.4091 & -1.8027 \\ 1.3391 & 0.2314 \end{pmatrix} \end{matrix}$$

Das Pivot $Piv = 0.4091$ sei z.B. **nicht** akzeptabel. Dessen Elimination wird auf einen späteren Zeitpunkt verschoben (verspätet).

$$U_{10} = F_{10} = \begin{matrix} \\ 5 \\ 10 \end{matrix} \begin{matrix} \begin{matrix} 5 & 10 \end{matrix} \\ \begin{pmatrix} 0.4091 & -1.8027 \\ 1.3391 & 0.2314 \end{pmatrix} \end{matrix}$$

mit $\mathcal{L}'_{10} = \{5, 10\}$.

Eigentlich ist der multifrontale Prozess jetzt zu Ende. Es ist die Matrix U_{10} mit $\mathcal{L}'_{10} = \{5, 10\}$ übrig geblieben. Hier kann man bliebige Gauß-Varianten anwenden, Z.B. full-Pivoting:

$$\begin{matrix} \\ 5 \\ 10 \end{matrix} \begin{matrix} \begin{matrix} 5 & 10 \end{matrix} \\ \begin{pmatrix} 0.4091 & -1.8027 \\ 1.3391 & 0.2314 \end{pmatrix} \end{matrix} \xrightarrow[\text{die Spalten}]{\text{permutiere}} \begin{matrix} \\ 5 \\ 10 \end{matrix} \begin{matrix} \begin{matrix} 10 & 5 \end{matrix} \\ \begin{pmatrix} -1.8027 & 0.4091 \\ 0.2314 & 1.3391 \end{pmatrix} \end{matrix} \xrightarrow{\text{eliminiere}}$$

$$\begin{matrix} \\ 5 \\ 10 \end{matrix} \begin{matrix} \begin{matrix} \\ \end{matrix} \\ \begin{pmatrix} 1 & 0 \\ -0.1284 & 1 \end{pmatrix} \end{matrix} \begin{pmatrix} -1.8027 & 0 \\ 0 & 1.3916 \end{pmatrix} \begin{matrix} \begin{matrix} 10 & 5 \end{matrix} \\ \begin{pmatrix} 1 & -0.2269 \\ 0 & 1 \end{pmatrix} \end{matrix}$$

Die Spalte(9) von L: $\begin{matrix} 5 \\ 10 \end{matrix} \begin{pmatrix} 1 \\ -0.1284 \end{pmatrix}$, die Zeile(9) von U: $\begin{matrix} 10 & 5 \\ \end{matrix} \begin{pmatrix} 1 & -0.2269 \end{pmatrix}$, $\pi(9) = 5,\ \psi(9) = 10,\ D(9) = -1.8027$ und $\pi(10) = 10,\ \psi(10) = 5,\ D(10) = 1.3961$.

Als Permutationen haben wir also:

$$\pi = (1, 2, 3, 4, 6, 8, 9, 7, 5, 10),\ \psi = (1, 2, 3, 4, 6, 8, 9, 7, 10, 5)$$

und deren Inverse:

$$\pi^{-1} = (1, 2, 3, 4, 9, 5, 8, 6, 7, 10),\ \psi^{-1} = (1, 2, 3, 4, 10, 5, 8, 6, 7, 9)$$

Damit wir die endgültige Zerlegung $A_e(\pi, \psi) = LDU$ erhalten, müssen wir noch π^{-1} bzw. ψ^{-1} auf die bisher berechneten L bzw. U anwenden: (Grün sind die neuen, endgültigen Indizes)

- Die Spalten von L:

$$(1): \begin{matrix} 1 \\ 3 \\ 5 \end{matrix} \begin{pmatrix} 1 \\ 0.7 \\ -0.5 \end{pmatrix} \begin{matrix} 1 \\ 3 \\ 9 \end{matrix}, \quad (2): \begin{matrix} 2 \\ 3 \\ 10 \end{matrix} \begin{pmatrix} 1 \\ 0.0 \\ 0.3 \end{pmatrix} \begin{matrix} 2 \\ 3 \\ 10 \end{matrix}, \quad (3): \begin{matrix} 3 \\ 5 \\ 10 \end{matrix} \begin{pmatrix} 1 \\ 0.9091 \\ -0.8409 \end{pmatrix} \begin{matrix} 3 \\ 9 \\ 10 \end{matrix},$$

$$(4): \begin{matrix} 4 \\ 5 \\ 10 \end{matrix} \begin{pmatrix} 1 \\ -0.9 \\ -0.5 \end{pmatrix} \begin{matrix} 4 \\ 9 \\ 10 \end{matrix}, \quad (5): \begin{matrix} 6 \\ 7 \\ 9 \end{matrix} \begin{pmatrix} 1 \\ 0.1 \\ -0.7 \end{pmatrix} \begin{matrix} 5 \\ 8 \\ 7 \end{matrix}, \quad (6): \begin{matrix} 8 \\ 9 \\ 10 \end{matrix} \begin{pmatrix} 1 \\ 0.0 \\ 0.7 \end{pmatrix} \begin{matrix} 6 \\ 7 \\ 10 \end{matrix},$$

$$(7):\ \begin{matrix}7\\9\\10\end{matrix}\begin{pmatrix}0.0201\\1\\0.3758\end{pmatrix}\begin{matrix}8\\7\\10\end{matrix},\ (8):\ \begin{matrix}7\\10\end{matrix}\begin{pmatrix}1\\-0.2928\end{pmatrix}\begin{matrix}8\\10\end{matrix},\ (9):\ \begin{matrix}5\\10\end{matrix}\begin{pmatrix}1\\-0.1284\end{pmatrix}\begin{matrix}9\\10\end{matrix}.$$

- Die Zeilen von U:

$$\begin{matrix}1&3&5\\1&3&10\end{matrix}(1):\begin{pmatrix}1&0.8&0.2\end{pmatrix},\ \begin{matrix}2&3&10\\1&3&9\end{matrix}(2):\begin{pmatrix}1&0.9&0.2\end{pmatrix},\ \begin{matrix}3&5&10\\3&10&9\end{matrix}(3):\begin{pmatrix}1&1.7273&0.6818\end{pmatrix},$$
$$\begin{matrix}4&5&10\\4&10&9\end{matrix}(4):\begin{pmatrix}1&0.0&-0.7\end{pmatrix},\ \begin{matrix}6&7&9\\5&8&7\end{matrix}(5):\begin{pmatrix}1&-0.3&0.7\end{pmatrix},\ \begin{matrix}8&9&10\\6&7&9\end{matrix}(6):\begin{pmatrix}1&-0.8&0.6\end{pmatrix},$$
$$\begin{matrix}7&9&10\\8&7&9\end{matrix}(7):\begin{pmatrix}0.5302&1&0.3356\end{pmatrix},\ \begin{matrix}7&10\\8&9\end{matrix}(8):\begin{pmatrix}1&-0.0099\end{pmatrix},\ \begin{matrix}10&5\\9&10\end{matrix}(9):\begin{pmatrix}1&-0.2269\end{pmatrix}.$$

Die endgültige Zerlegung ist $A_e(\pi,\psi) = LDU$ mit:

$$A(\pi,\psi) = \begin{pmatrix}
1 & 0 & 0.8 & 0 & 0 & 0 & 0 & 0 & 0 & 0.2\\
0 & 1 & 0.9 & 0 & 0 & 0 & 0 & 0 & 0.2 & 0\\
0.7 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0.3 & 0.9\\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & -0.7 & 0\\
0 & 0 & 0 & 0 & 1 & 0 & 0.7 & -0.3 & 0 & 0\\
0 & 0 & 0 & 0 & 0 & 1 & -0.8 & 0 & 0.6 & 0\\
0 & 0 & 0 & 0 & -0.7 & 0 & 1 & 1 & 0.5 & 0\\
0 & 0 & 0 & 0 & 0.1 & 0 & 0.1 & 1 & 0 & 0\\
-0.5 & 0 & 0 & -0.9 & 0 & 0 & 0 & 0 & -0.9 & 1\\
0 & 0.3 & -0.1 & -0.5 & 0 & 0.7 & 0 & 0 & 1 & 0.7
\end{pmatrix},$$

$$L = \begin{pmatrix}
1 & & & & & & & & & \\
0 & 1 & & & & & & & & \\
0.7 & 0 & 1 & & & & & & & \\
0 & 0 & 0 & 1 & & & & & & \\
0 & 0 & 0 & 0 & 1 & & & & & \\
0 & 0 & 0 & 0 & 0 & 1 & & & & \\
0 & 0 & 0 & 0 & -0.7 & 0 & 1 & & & \\
0 & 0 & 0 & 0 & 0.1 & 0 & 0.0201 & 1 & & \\
-0.5 & 0 & 0.9091 & -0.9 & 0 & 0 & 0 & 0 & 1 & \\
0 & 0.3 & -0.8409 & -0.5 & 0 & 0.7 & 0.3758 & -0.2928 & -0.1284 & 1
\end{pmatrix},$$

$D = \mathrm{diag}(1,1,0.44,1,1,1,1.49,1.0141,-1.8027,1.3916)$ und

$$U = \begin{pmatrix} 1 & 0 & 0.8 & 0 & 0 & 0 & 0 & 0 & 0 & 0.2 \\ & 1 & 0.9 & 0 & 0 & 0 & 0 & 0 & 0.2 & 0 \\ & & 1 & 0 & 0 & 0 & 0 & 0 & 0.6818 & 1.7273 \\ & & & 1 & 0 & 0 & 0 & 0 & -0.7 & 0 \\ & & & & 1 & 0 & 0.7 & -0.3 & 0 & 0 \\ & & & & & 1 & -0.8 & 0 & 0.6 & 0 \\ & & & & & & 1 & 0.5302 & 0.3356 & 0 \\ & & & & & & & 1 & -0.0099 & 0 \\ & & & & & & & & 1 & -0.2269 \\ & & & & & & & & & 1 \end{pmatrix}.$$

7.2 Dropping wird eingeführt

Prinzipiell kann man aus jeder LDU Zerlegung, d.h. jede Gauß Variante, eine ILU machen. Die multifrontale Methode ist sehr effizient für direkte LDU-Zerlegungen von Matrizen mit symmetrischen Struktur. Durch das Software Packet UMFPACK[6] ([22]), das ab MATLAB 7 die Standard-Methode für die LDU-Zerlegung (dünn besetzter Matrizen) mit beidseitiger[7] Pivotisierung ist, ist dessen Effizienz auch auf die Matrizen mit unsymmetrischen Struktur erweitert worden. Die effiziente Parallelisierung von *LDU*-Zerlegungen für dünn besetzte Matrizen, die immer ein schwieriges Problem ist, ist dank der multifrontalen Methode leichter anzupacken.

Das waren die Gründe warum ich die multifrontale Methode ausgewählt habe[8]. Außerdem, gab es meine Recherchen nach nichts derartiges, welches den multifrontalen Ansatz auf eine ILU überträgt.

Man muss sich im Klaren sein, dass es von vornherein Verträglichkeitprobleme gibt. Z.B. da ich die frontalen Matrizen im sparse Format speichere, werden die BLAS-2 und BLAS-3 Routinen ([1]), die bei der direkten Variante anwendbar sind, nicht anwendbar. Ein anderes Problem ist explizit mit dem inverse based dropping verbunden. Es verhindert nämlich “tree parallelism“, d.h. Knoten, die sich in disjunkten Teilbäumen des Eliminationsbaums befinden, können nicht ganz “sauber“ unabhängig voneinander prozessiert werden. Dies wird später eingehend erläutert.

Das übliche dropping kann ohne viele Probleme in die multifrontale Methode eingeführt werden. Das Inverse based dropping erfordert dagegen einige zusätzliche Arbeit, denn die Einbeziehung des condition estimators für das inverse based dropping ist nicht trivial. Weil die Blätter des Eliminationsbaums $T[A]$ eine abweichende Behandlung verdienen, habe ich den condition estimator Algorithmus

[6] unsymmetric-pattern multifrontal method for sparse LU factorization

[7] Bei einseitiger Pivotisierung ist meines Wissens nach der schnellste Code der von Gilbert und Peierls [40]. Der ist längst ein Bestandteil von MATLAB

[8] abgesehen von ihre Schönheit

(siehe Seite 110) so modifiziert, dass bei den Blättern kein dropping angewandt wird. Der resultierende Algorithmus ist:

Algorithmus 7.1 $\|\cdot\|_1$ cond. est. für die Zeilen der L^{-1} bzw. Spalten der U^{-1}

$v^L = (0,0,\ldots,0)^T$, $CondEst^L = (0,0,\ldots,0)^T$
$v^U = (0,0,\ldots,0)$, $CondEst^U = (0,0,\ldots,0)$
for $k = 1 : n$ **do**
  {Lokal}
  if k Blatt im $T[A]$ **then**
    assert$(v_k^L == 0)$
    assert$(v_k^U == 0)$
    setze $x_+^L = +1$, $x_-^L = -1$
    setze $x_+^U = +1$, $x_-^U = -1$
    wende keine dropping auf $\tilde{l}_{k+1:n,k}$ an
    wende keine dropping auf $\tilde{u}_{k,k+1:n}$ an
  else
    setze $x_+^L = +1 - v_k^L$, $x_-^L = -1 - v_k^L$
    setze $x_+^U = +1 - v_k^U$, $x_-^U = -1 - v_k^U$
    $CondEst^L(k) = \max\left(\left|x_-^L\right|, \left|x_+^L\right|\right)$ {es gilt: $CondEst^L(k) \geq 1$}
    $CondEst^U(k) = \max\left(\left|x_-^U\right|, \left|x_+^U\right|\right)$ {es gilt: $CondEst^U(k) \geq 1$}
    wende dropping auf $\tilde{l}_{k+1:n,k}$ an
    wende dropping auf $\tilde{u}_{k,k+1:n}$ an
  end if
  {Global}
  setze $Q^L = \text{Struct}\left(\tilde{l}_{k+1,k}, \ldots, \tilde{l}_{n,k}\right)$
  setze $Q^U = \text{Struct}(\tilde{u}_{k,k+1}, \ldots, \tilde{u}_{k,n})$
  setze $\nu_+^L = \left\| v_{Q^L}^L + x_+^L \tilde{l}_{Q^L,k} \right\|_1$, $\nu_-^L = \left\| v_{Q^L}^L + x_-^L \tilde{l}_{Q^L,k} \right\|_1$
  setze $\nu_+^U = \left\| v_{Q^U}^U + x_+^U \tilde{u}_{k,Q^U} \right\|_1$, $\nu_-^U = \left\| v_{Q^U}^U + x_-^U \tilde{u}_{k,Q^U} \right\|_1$
  if $\nu_+^L \geq \nu_-^L$ **then**
    $v_{Q^L}^L = v_{Q^L}^L + x_+^L \tilde{l}_{Q^L,k}$
  else
    $v_{Q^L}^L = v_{Q^L}^L + x_-^L \tilde{l}_{Q^L,k}$
  end if
  if $\nu_+^U \geq \nu_-^U$ **then**
    $v_{Q^U}^U = v_{Q^U}^U + x_+^U \tilde{u}_{k,Q^U}$
  else
    $v_{Q^U}^U = v_{Q^U}^U + x_-^U \tilde{u}_{k,Q^U}$
  end if
end for

Man beachte die Zeilen 21 und 22. Der Struct Operator wird angewandt, nachdem Dropping angewandt wurde (Zeilen 17,18). Das macht die Aufdatierungen der globalen Vektoren v^L und v^U leichter.

Im Schritt k, für k **nicht** Blatt, werden alle $\tilde{l}_{j,k}$ bzw. $\tilde{u}_{k,j}$ verworfen mit:

$$\left|\tilde{l}_{j,k}\right| \cdot CondEst^L(k) \leq \varepsilon \quad \text{bzw.} \quad |\tilde{u}_{k,j}| \cdot CondEst^U(k) \leq \varepsilon$$

Um also, das inverse based dropping für unser 10×10 Beispiel einzuführen, brauchen wir ein ε, z.B. $\varepsilon = 0.9$.

Benötigt wird auch eine Pivotisierungsstrategie, d.h. ein Kriterium, dass ein totally-summed Knoten erfüllen muss, um als Pivot akzeptabel zu sein. Diese Kriterium wird mit großem Gewicht zur Performance des ILU beitragen. Vor allem muss es schnell nachprüfbar sein. Ich werde bei diesem Beispiel keine explizite Pivotisierungsstrategie anwenden. In erster Linie geht es mir darum zu verdeutlichen, wie die Methode funktioniert. Explizite Pivotisierungsstrategien sind leicht nachrüstbar.

Um die Berechnungen übersichtlich zu halten, werde ich in diesem Beispiel die S-Variante als Schur-Update Variante benutzten (siehe Gleichung 6.6).

Fangen wir an:

$n = 10$, $\varepsilon = 0.9$, $v^L = v^U = zeros(1, n)$, $CE^L = CE^U = ones(1, n)$. Um die Beschreibung nicht unnötig aufzublähen, werden nur relevante Einträge von v^L und v^U ausgegeben.

- Knoten $k = 1$: Er ist ein Blatt (keine Update-Matrizen aus früheren Schritten, dropping wird nicht angewandt)

$$F_1 = \begin{array}{c} \\ 1 \\ 3 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 3 & 5 \end{array} \\ \begin{pmatrix} 1 & 0.8 & 0.2 \\ 0.7 & 0 & 0 \\ -0.5 & 0 & 0 \end{pmatrix} \end{array}$$

  Der Knoten 1 ist totally summed und ist bereit für die Elimination. Das Pivot $Piv = 1$ sei z.B. akzeptabel. Die Spalte(1) von L: $\begin{array}{c} 1 \\ 3 \\ 5 \end{array} \begin{pmatrix} 1 \\ 0.7 \\ -0.5 \end{pmatrix}$, die Zeile(1) von U: $\begin{array}{ccc} 1 & 3 & 5 \end{array} \begin{pmatrix} 1 & 0.8 & 0.2 \end{pmatrix}$, die Update-Matrix: $U_1 = \begin{array}{c} \\ 3 \\ 5 \end{array} \begin{array}{c} \begin{array}{cc} 3 & 5 \end{array} \\ \begin{pmatrix} -0.56 & -0.14 \\ 0.4 & 0.1 \end{pmatrix} \end{array}$

  und zuletzt $\pi(1) = 1$, $\psi(1) = 1$, $D(1) = 1.0$.

  Datiere v^L und v^U auf:

  $Q^L = \{3, 5\}$, $Q^U = \{3, 5\}$, $x_+^L = 1$, $x_-^L = -1$, $x_+^U = 1$, $x_-^U = -1$.

$$\nu_{\pm}^{L} = \left\| v_{Q^L}^{L} + x_{\pm}^{L} \tilde{l}_{Q^L,k} \right\|_1 = \|() \pm 1 \cdot \begin{array}{c} \begin{array}{cc} 3 & 5 \end{array} \\ \begin{pmatrix} 0.7 & -0.5 \end{pmatrix} \end{array} \|_1 = 1.2$$

$$\nu_{\pm}^{U} = \left\| v_{Q^U}^{U} + x_{\pm}^{U} \tilde{u}_{k,Q^U} \right\|_1 = \|() \pm 1 \cdot \begin{array}{c} \begin{array}{cc} 3 & 5 \end{array} \\ \begin{pmatrix} 0.8 & 0.2 \end{pmatrix} \end{array} \|_1 = 1.0. \text{ Daher ist:}$$

$$v^L = v^L + x_{+}^{L} \tilde{l}_{Q^L,k} = () + 1 \cdot \begin{array}{c} \begin{array}{cc} 3 & 5 \end{array} \\ \begin{pmatrix} 0.7 & -0.5 \end{pmatrix} \end{array} = \begin{array}{c} \begin{array}{cc} 3 & 5 \end{array} \\ \begin{pmatrix} 0.7 & -0.5 \end{pmatrix} \end{array}$$

$$v^U = v^U + x_{+}^{U} \tilde{u}_{k,Q^U} = () + 1 \cdot \begin{array}{c} \begin{array}{cc} 3 & 5 \end{array} \\ \begin{pmatrix} 0.8 & 0.2 \end{pmatrix} \end{array} = \begin{array}{c} \begin{array}{cc} 3 & 5 \end{array} \\ \begin{pmatrix} 0.8 & 0.2 \end{pmatrix} \end{array}$$

- Knoten $k = 2$: Er ist ein Blatt (keine Update-Matrizen aus früheren Schritten, dropping wird nicht angewandt)

$$F_2 = \begin{array}{c} \begin{array}{ccc} 2 & 3 & 10 \end{array} \\ \begin{array}{c} 2 \\ 10 \end{array} \begin{pmatrix} 1 & 0.9 & 0.2 \\ 0.3 & 0 & 0 \end{pmatrix} \end{array}$$

  Der Knoten 2 ist totally summed und ist bereit für die Elimination. Das Pivot $Piv = 1$ sei z.B. akzeptabel. Die Spalte(2) von L: $\begin{array}{c} 2 \\ 10 \end{array} \begin{pmatrix} 1 \\ 0.3 \end{pmatrix}$, die Zeile(2) von U: $\begin{array}{c} \begin{array}{ccc} 2 & 3 & 10 \end{array} \\ \begin{pmatrix} 1 & 0.9 & 0.2 \end{pmatrix} \end{array}$, die Update-Matrix: $U_2 = \begin{array}{c} \begin{array}{cc} 3 & 10 \end{array} \\ \begin{array}{c} 10 \end{array} \begin{pmatrix} -0.27 & -0.06 \end{pmatrix} \end{array}$ und zuletzt $\pi(2) = 2$, $\psi(2) = 2$, $D(2) = 1.0$.

  Datiere v^L und v^U auf:

  $Q^L = \{10\}$, $Q^U = \{3, 10\}$, $x_{+}^{L} = 1$, $x_{-}^{L} = -1$, $x_{+}^{U} = 1$, $x_{-}^{U} = -1$.

$$\nu_{\pm}^{L} = \|() \pm 1 \cdot \begin{array}{c} 10 \\ \begin{pmatrix} 0.3 \end{pmatrix} \end{array} \|_1 = 0.3$$

$$\nu_{+}^{U} = \|\begin{array}{c} 3 \\ \begin{pmatrix} 0.8 \end{pmatrix} \end{array} + 1 \cdot \begin{array}{c} \begin{array}{cc} 3 & 10 \end{array} \\ \begin{pmatrix} 0.9 & 0.2 \end{pmatrix} \end{array} \|_1 = 1.9$$

$$\nu_{-}^{U} = \|\begin{array}{c} 3 \\ \begin{pmatrix} 0.8 \end{pmatrix} \end{array} - 1 \cdot \begin{array}{c} \begin{array}{cc} 3 & 10 \end{array} \\ \begin{pmatrix} 0.9 & 0.2 \end{pmatrix} \end{array} \|_1 = 0.3$$

  Daher ist:

$$v^L = v^L + x_{+}^{L} \tilde{l}_{Q^L,k} = \begin{array}{c} \begin{array}{cc} 3 & 5 \end{array} \\ \begin{pmatrix} 0.7 & -0.5 \end{pmatrix} \end{array} + 1 \cdot \begin{array}{c} 10 \\ \begin{pmatrix} 0.3 \end{pmatrix} \end{array} = \begin{array}{c} \begin{array}{ccc} 3 & 5 & 10 \end{array} \\ \begin{pmatrix} 0.7 & -0.5 & 0.3 \end{pmatrix} \end{array}$$

$$v^U = v^U + x_{+}^{U} \tilde{u}_{k,Q^U} = \begin{array}{c} \begin{array}{cc} 3 & 5 \end{array} \\ \begin{pmatrix} 0.8 & 0.2 \end{pmatrix} \end{array} + 1 \cdot \begin{array}{c} \begin{array}{cc} 3 & 10 \end{array} \\ \begin{pmatrix} 0.9 & 0.2 \end{pmatrix} \end{array} = \begin{array}{c} \begin{array}{ccc} 3 & 5 & 10 \end{array} \\ \begin{pmatrix} 1.7 & 0.2 & 0.2 \end{pmatrix} \end{array}$$

- Knoten $k = 3$: Er besitzt die Knoten 1 und 2 als Söhne.

$$F_3 = \begin{array}{c} \begin{array}{ccc} 3 & 5 & 10 \end{array} \\ \begin{array}{c} 3 \\ 10 \end{array} \begin{pmatrix} 1 & 0.9 & 0.3 \\ -0.1 & 0 & 0 \end{pmatrix} \end{array} \oplus U_1 \oplus U_2 = \begin{array}{c} \begin{array}{ccc} 3 & 5 & 10 \end{array} \\ \begin{array}{c} 3 \\ 5 \\ 10 \end{array} \begin{pmatrix} 0.44 & 0.76 & 0.3 \\ 0.4 & 0.1 & 0.0 \\ -0.37 & 0.0 & -0.06 \end{pmatrix} \end{array}$$

Der Knoten 3 ist totally summed und ist bereit für die Elimination. Das Pivot $Piv = 0.44$ sei z.B. **nicht** akzeptabel. Dessen Elimination wird auf einen späteren Zeitpunkt verschoben (verspätet).

$$U_3 = F_3 = \begin{array}{c} \\ 3 \\ 5 \\ 10 \end{array} \begin{array}{c} \begin{array}{ccc} 3 & 5 & 10 \end{array} \\ \begin{pmatrix} 0.44 & 0.76 & 0.3 \\ 0.4 & 0.1 & 0.0 \\ -0.37 & 0.0 & -0.06 \end{pmatrix} \end{array}$$

Verspätete Knoten werden rot gefärbt. Bei v^L und v^U ändert sich nichts:

$$v^L = \begin{array}{c} \begin{array}{ccc} 3 & 5 & 10 \end{array} \\ \begin{pmatrix} 0.7 & -0.5 & 0.3 \end{pmatrix} \end{array}, \; v^U = \begin{array}{c} \begin{array}{ccc} 3 & 5 & 10 \end{array} \\ \begin{pmatrix} 1.7 & 0.2 & 0.2 \end{pmatrix} \end{array}$$

- Knoten $k = 4$: Er ist ein Blatt (keine Update-Matrizen aus früheren Schritten, dropping wird nicht angewandt)

$$F_4 = \begin{array}{c} \\ 4 \\ 5 \\ 10 \end{array} \begin{array}{c} \begin{array}{cc} 4 & 10 \end{array} \\ \begin{pmatrix} 1 & -0.7 \\ -0.9 & 0 \\ -0.5 & 0 \end{pmatrix} \end{array}$$

Der Knoten 4 ist totally summed und ist bereit für die Elimination. Das Pivot $Piv = 1$ sei z.B. akzeptabel. Die Spalte(3) von L: $\begin{array}{c} 4 \\ 5 \\ 10 \end{array} \begin{pmatrix} 1 \\ -0.9 \\ -0.5 \end{pmatrix}$,

die Zeile(3) von U: $\begin{array}{c} \begin{array}{cc} 4 & 10 \end{array} \\ \begin{pmatrix} 1 & -0.7 \end{pmatrix} \end{array}$, die Update-Matrix: $U_4 = \begin{array}{c} \\ 5 \\ 10 \end{array} \begin{array}{c} \begin{array}{c} 10 \end{array} \\ \begin{pmatrix} -0.63 \\ -0.35 \end{pmatrix} \end{array}$

und zuletzt $\pi(3) = 4$, $\psi(3) = 4$, $D(3) = 1.0$.

Datiere v^L und v^U auf:

$Q^L = \{5, 10\}$, $Q^U = \{10\}$, $x^L_+ = 1$, $x^L_- = -1$, $x^U_+ = 1$, $x^U_- = -1$.

$$\nu^L_+ = \| \begin{array}{c} \begin{array}{cc} 5 & 10 \end{array} \\ \begin{pmatrix} -0.5 & 0.3 \end{pmatrix} \end{array} + 1 \cdot \begin{array}{c} \begin{array}{cc} 5 & 10 \end{array} \\ \begin{pmatrix} -0.9 & -0.5 \end{pmatrix} \end{array} \|_1 = 1.6$$

$$\nu^L_- = \| \begin{array}{c} \begin{array}{cc} 5 & 10 \end{array} \\ \begin{pmatrix} -0.5 & 0.3 \end{pmatrix} \end{array} - 1 \cdot \begin{array}{c} \begin{array}{cc} 5 & 10 \end{array} \\ \begin{pmatrix} -0.9 & -0.5 \end{pmatrix} \end{array} \|_1 = 1.2$$

$$\nu^U_+ = \| \begin{array}{c} \begin{array}{c} 10 \end{array} \\ \begin{pmatrix} 0.2 \end{pmatrix} \end{array} + 1 \cdot \begin{array}{c} \begin{array}{c} 10 \end{array} \\ \begin{pmatrix} -0.7 \end{pmatrix} \end{array} \|_1 = 0.5$$

$$\nu^U_- = \| \begin{array}{c} \begin{array}{c} 10 \end{array} \\ \begin{pmatrix} 0.2 \end{pmatrix} \end{array} - 1 \cdot \begin{array}{c} \begin{array}{c} 10 \end{array} \\ \begin{pmatrix} -0.7 \end{pmatrix} \end{array} \|_1 = 0.9$$

Daher ist:

$$v^L = v^L + x^L_+ \tilde{l}_{Q^L,k} = \begin{array}{c} \begin{array}{ccc} 3 & 5 & 10 \end{array} \\ \begin{pmatrix} 0.7 & -0.5 & 0.3 \end{pmatrix} \end{array} + 1 \cdot \begin{array}{c} \begin{array}{cc} 5 & 10 \end{array} \\ \begin{pmatrix} -0.9 & -0.5 \end{pmatrix} \end{array} = \begin{array}{c} \begin{array}{ccc} 3 & 5 & 10 \end{array} \\ \begin{pmatrix} 0.7 & -1.4 & -0.2 \end{pmatrix} \end{array}$$

$$v^U = v^U + x^U_- \tilde{u}_{k,Q^U} = \begin{array}{c} \begin{array}{ccc} 3 & 5 & 10 \end{array} \\ \begin{pmatrix} 1.7 & 0.2 & 0.2 \end{pmatrix} \end{array} - 1 \cdot \begin{array}{c} \begin{array}{c} 10 \end{array} \\ \begin{pmatrix} -0.7 \end{pmatrix} \end{array} = \begin{array}{c} \begin{array}{ccc} 3 & 5 & 10 \end{array} \\ \begin{pmatrix} 1.7 & 0.2 & 0.9 \end{pmatrix} \end{array}$$

- Knoten $k = 5$: Er besitzt die Knoten 3 und 4 als Söhne.

$$F_5 = \begin{matrix} \\ 5 \\ 10 \end{matrix}\!\!\begin{matrix} \begin{matrix} 5 & 10 \end{matrix} \\ \begin{pmatrix} 1 & -0.9 \\ 0.7 & 0 \end{pmatrix} \end{matrix} \oplus U_3 \oplus U_4 = \begin{matrix} \\ 3 \\ 5 \\ 10 \end{matrix}\!\!\begin{matrix} \begin{matrix} 3 & 5 & 10 \end{matrix} \\ \begin{pmatrix} 0.44 & 0.76 & 0.3 \\ 0.4 & 1.1 & -1.53 \\ -0.37 & 0.7 & -0.41 \end{pmatrix} \end{matrix}$$

  Der Knoten 5 ist totally summed und ist bereit für die Elimination. Das Pivot $Piv = 1.1$ sei z.B. akzeptabel. Die Spalte(4) von L: $\begin{matrix} \\ 3 \\ 5 \\ 10 \end{matrix}\!\!\begin{matrix} \\ \begin{pmatrix} 0.6909 \\ 1 \\ 0.6364 \end{pmatrix} \end{matrix}$, die Zeile(4) von U: $\begin{matrix} \begin{matrix} 3 & 5 & 10 \end{matrix} \\ \begin{pmatrix} 0.3636 & 1 & -1.3909 \end{pmatrix} \end{matrix}$ und zuletzt $\pi(4) = 5$, $\psi(4) = 5$, $D(4) = 1.1$.

  Berechne Condition Estimators: $x_+^L = 1 - v_k^L = 1 - (-1.4) = 2.4$, $x_-^L = -1 - v_k^L = -1 - (-1.4) = 0.4$, $x_+^U = 1 - v_k^U = 1 - 0.2 = 0.8$, $x_-^U = -1 - v_k^U = -1 - 0.2 = -1.2$.

  $CE_k^L = \max(|x_+^L|, |x_-^L|) = 2.4$, $CE_k^U = \max(|x_+^U|, |x_-^U|) = 1.2$

  Es werden alle $\left|\tilde{l}_{j,k}\right| \leq \frac{\varepsilon}{CE_k^L} = \frac{0.9}{2.4} = 0.375$ und alle $|\tilde{u}_{k,j}| \leq \frac{\varepsilon}{CE_k^U} = \frac{0.9}{1.2} = 0.75$ gedropped.

  Wende dropping an: Die Spalte(4) von L: $\begin{matrix} \\ 3 \\ 5 \\ 10 \end{matrix}\!\!\begin{matrix} \\ \begin{pmatrix} 0.6909 \\ 1 \\ 0.6364 \end{pmatrix} \end{matrix}$, die Zeile(4) von U: $\begin{matrix} \begin{matrix} 5 & 10 \end{matrix} \\ \begin{pmatrix} 1 & -1.3909 \end{pmatrix} \end{matrix}$, die Update-Matrix: $U_5 = \begin{matrix} \\ 3 \\ 10 \end{matrix}\!\!\begin{matrix} \begin{matrix} 3 & 10 \end{matrix} \\ \begin{pmatrix} 0.44 & 1.3571 \\ -0.37 & 0.5636 \end{pmatrix} \end{matrix}$.

  Datiere v^L und v^U auf: $Q^L = \{3, 10\}$, $Q^U = \{10\}$,

$$\nu_+^L = \left\| \begin{matrix} \begin{matrix} 3 & 10 \end{matrix} \\ \begin{pmatrix} 0.7 & -0.2 \end{pmatrix} \end{matrix} + (2.4) \cdot \begin{matrix} \begin{matrix} 3 & 10 \end{matrix} \\ \begin{pmatrix} 0.6909 & 0.6364 \end{pmatrix} \end{matrix} \right\|_1 = 3.6855$$

$$\nu_-^L = \left\| \begin{matrix} \begin{matrix} 3 & 10 \end{matrix} \\ \begin{pmatrix} 0.7 & -0.2 \end{pmatrix} \end{matrix} + (0.4) \cdot \begin{matrix} \begin{matrix} 3 & 10 \end{matrix} \\ \begin{pmatrix} 0.6909 & 0.6364 \end{pmatrix} \end{matrix} \right\|_1 = 1.0309$$

$$\nu_+^U = \left\| \begin{matrix} \begin{matrix} 10 \end{matrix} \\ \begin{pmatrix} 0.9 \end{pmatrix} \end{matrix} + 0.8 \cdot \begin{matrix} \begin{matrix} 10 \end{matrix} \\ \begin{pmatrix} -1.3909 \end{pmatrix} \end{matrix} \right\|_1 = 0.2127$$

$$\nu_-^U = \left\| \begin{matrix} \begin{matrix} 10 \end{matrix} \\ \begin{pmatrix} 0.9 \end{pmatrix} \end{matrix} + (-1.2) \cdot \begin{matrix} \begin{matrix} 10 \end{matrix} \\ \begin{pmatrix} -1.3909 \end{pmatrix} \end{matrix} \right\|_1 = 2.5691$$

  Daher ist:

$$v^L = v^L + x_+^L \tilde{l}_{Q^L,k} = \begin{matrix} \begin{matrix} 3 & 10 \end{matrix} \\ \begin{pmatrix} 0.7 & -0.2 \end{pmatrix} \end{matrix} + 2.4 \cdot \begin{matrix} \begin{matrix} 3 & 10 \end{matrix} \\ \begin{pmatrix} 0.6909 & 0.6364 \end{pmatrix} \end{matrix} = \begin{matrix} \begin{matrix} 3 & 10 \end{matrix} \\ \begin{pmatrix} 2.3582 & 1.3274 \end{pmatrix} \end{matrix}$$

$$v^U = v^U + x_-^U \tilde{u}_{k,Q^U} = \begin{matrix} \begin{matrix} 3 & 10 \end{matrix} \\ \begin{pmatrix} 1.7 & 0.9 \end{pmatrix} \end{matrix} + (-1.2) \cdot \begin{matrix} \begin{matrix} 10 \end{matrix} \\ \begin{pmatrix} -1.3909 \end{pmatrix} \end{matrix} = \begin{matrix} \begin{matrix} 3 & 10 \end{matrix} \\ \begin{pmatrix} 1.7 & 2.5691 \end{pmatrix} \end{matrix}$$

Da der Knoten 3 totally summed ist, wäre es möglich, ihn aus U_5 zu eliminieren (was ich nicht machen werde).

- Knoten $k = 6$: Er ist ein Blatt (keine Update-Matrizen aus früheren Schritten, dropping wird nicht angewandt)

$$F_6 = \begin{array}{c} \\ 6 \\ 7 \\ 9 \end{array} \begin{array}{c} \begin{array}{ccc} 6 & 7 & 9 \end{array} \\ \begin{pmatrix} 1 & -0.3 & 0.7 \\ 0.1 & 0 & 0 \\ -0.7 & 0 & 0 \end{pmatrix} \end{array}$$

Der Knoten 6 ist totally summed und ist bereit für die Elimination. Das Pivot $Piv = 1$ sei z.B. akzeptabel. Die Spalte(5) von L: $\begin{array}{c} 6 \\ 7 \\ 9 \end{array} \begin{pmatrix} 1 \\ 0.1 \\ -0.7 \end{pmatrix}$, die Zeile(5) von U: $\begin{array}{c} 6 \quad 7 \quad 9 \\ (\ 1 \quad -0.8 \quad 0.7\ ) \end{array}$, die Update-Matrix: $U_6 = \begin{array}{c} \\ 7 \\ 9 \end{array} \begin{array}{c} \begin{array}{cc} 7 & 9 \end{array} \\ \begin{pmatrix} 0.03 & -0.07 \\ -0.21 & 0.49 \end{pmatrix} \end{array}$ und zuletzt $\pi(5) = 6$, $\psi(5) = 6$, $D(5) = 1$.

Datiere v^L und v^U auf:

$Q^L = \{7, 9\}$, $Q^U = \{7, 9\}$, $x^L_+ = 1$, $x^L_- = -1$, $x^U_+ = 1$, $x^U_- = -1$.

$\nu^L_\pm = \|() \pm 1 \cdot \begin{array}{c} 7 \quad 9 \\ (\ 0.1 \quad -0.7\ ) \end{array} \|_1 = 0.8$

$\nu^U_\pm = \|() \pm 1 \cdot \begin{array}{c} 7 \quad 9 \\ (\ -0.3 \quad 0.7\ ) \end{array} \|_1 = 1.0.$

Daher ist:

$$v^L = \begin{array}{c} 3 \quad 10 \\ (\ 2.3582 \quad 1.3274\ ) \end{array} + 1 \cdot \begin{array}{c} 7 \quad 9 \\ (\ 0.1 \quad -0.7\ ) \end{array} = \begin{array}{c} 3 \quad 7 \quad 9 \quad 10 \\ (\ 2.3582 \quad 0.1 \quad -0.7 \quad 1.3274\ ) \end{array}$$

$$v^U = \begin{array}{c} 3 \quad 10 \\ (\ 1.7 \quad 2.5691\ ) \end{array} + 1 \cdot \begin{array}{c} 7 \quad 9 \\ (\ -0.3 \quad 0.7\ ) \end{array} = \begin{array}{c} 3 \quad 7 \quad 9 \quad 10 \\ (\ 1.7 \quad -0.3 \quad 0.7 \quad 2.5691\ ) \end{array}$$

- Knoten $k = 7$: Er besitzt den Knoten 6 als Sohn.

$$F_7 = \begin{array}{c} \\ 7 \\ 9 \end{array} \begin{array}{c} \begin{array}{cc} 7 & 9 \end{array} \\ \begin{pmatrix} 1 & 0.1 \\ 1 & 0 \end{pmatrix} \end{array} \oplus U_6 = \begin{array}{c} \\ 7 \\ 9 \end{array} \begin{array}{c} \begin{array}{cc} 7 & 9 \end{array} \\ \begin{pmatrix} 1.03 & 0.03 \\ 0.79 & 0.49 \end{pmatrix} \end{array}$$

Der Knoten 7 ist totally summed und ist bereit für die Elimination. Das Pivot $Piv = 1.03$ sei z.B. **nicht** akzeptabel. Dessen Elimination wird auf eine spätere Zeitpunkt verschoben (verspätet).

$$U_7 = F_7 = \begin{array}{c} \\ 7 \\ 9 \end{array} \begin{array}{c} \begin{array}{cc} 7 & 9 \end{array} \\ \begin{pmatrix} 1.03 & 0.03 \\ 0.79 & 0.49 \end{pmatrix} \end{array}$$

Bei v^L und v^U ändert sich nichts:

$$v^L = \begin{matrix} 3 & 7 & 9 & 10 \end{matrix}\begin{pmatrix} 2.3582 & 0.1 & -0.7 & 1.3274 \end{pmatrix},\ v^U = \begin{matrix} 3 & 7 & 9 & 10 \end{matrix}\begin{pmatrix} 1.7 & -0.3 & 0.7 & 2.5691 \end{pmatrix}$$

- Knoten $k = 8$: Er ist ein Blatt (keine Update-Matrizen aus früheren Schritten, dropping wird nicht angewandt)

$$F_8 = \begin{matrix} & 8 & 9 & 10 \\ 8 \\ 10 \end{matrix}\begin{pmatrix} 1 & -0.8 & 0.6 \\ 0.7 & 0 & 0 \end{pmatrix}$$

Der Knoten 8 ist totally summed und ist bereit für die Elimination. Das Pivot $Piv = 1$ sei z.B. akzeptabel. Die Spalte(6) von L: $\begin{matrix} 8 \\ 10 \end{matrix}\begin{pmatrix} 1 \\ 0.7 \end{pmatrix}$, die Zeile(6) von U: $\begin{matrix} 8 & 9 & 10 \end{matrix}\begin{pmatrix} 1 & -0.8 & 0.6 \end{pmatrix}$, die Update-Matrix: $U_8 = \begin{matrix} & 9 & 10 \\ 10 \end{matrix}\begin{pmatrix} 0.56 & -0.42 \end{pmatrix}$ und zuletzt $\pi(6) = 8$, $\psi(6) = 8$, $D(6) = 1$.

Datiere v^L und v^U auf:

$Q^L = \{10\}$, $Q^U = \{9, 10\}$, $x_+^L = 1$, $x_-^L = -1$, $x_+^U = 1$, $x_-^U = -1$.

$$\nu_+^L = \| \begin{matrix} 10 \\ \end{matrix}\begin{pmatrix} 1.3274 \end{pmatrix} + 1 \cdot \begin{matrix} 10 \\ \end{matrix}\begin{pmatrix} 0.7 \end{pmatrix} \|_1 = 2.0274$$

$$\nu_-^L = \| \begin{matrix} 10 \\ \end{matrix}\begin{pmatrix} 1.3274 \end{pmatrix} - 1 \cdot \begin{matrix} 10 \\ \end{matrix}\begin{pmatrix} 0.7 \end{pmatrix} \|_1 = 0.6274$$

$$\nu_+^U = \| \begin{matrix} 9 & 10 \end{matrix}\begin{pmatrix} 0.7 & 2.5691 \end{pmatrix} + 1 \cdot \begin{matrix} 9 & 10 \end{matrix}\begin{pmatrix} -0.8 & 0.6 \end{pmatrix} \|_1 = 3.2691$$

$$\nu_-^U = \| \begin{matrix} 9 & 10 \end{matrix}\begin{pmatrix} 0.7 & 2.5691 \end{pmatrix} - 1 \cdot \begin{matrix} 9 & 10 \end{matrix}\begin{pmatrix} -0.8 & 0.6 \end{pmatrix} \|_1 = 3.4691.$$

Daher ist:

$$v^L = (\ldots) + \begin{matrix} 10 \\ \end{matrix}\begin{pmatrix} 0.7 \end{pmatrix} = \begin{matrix} 3 & 7 & 9 & 10 \end{matrix}\begin{pmatrix} 2.3582 & 0.1 & -0.7 & 2.0274 \end{pmatrix}$$

$$v^U = (\ldots) - \begin{matrix} 9 & 10 \end{matrix}\begin{pmatrix} -0.8 & 0.6 \end{pmatrix} = \begin{matrix} 3 & 7 & 9 & 10 \end{matrix}\begin{pmatrix} 1.7 & -0.3 & 1.5 & 1.9691 \end{pmatrix}$$

- Knoten $k = 9$: Er besitzt die Knoten 7 und 8 als Söhne.

$$F_9 = \begin{matrix} & 9 & 10 \\ 9 \end{matrix}\begin{pmatrix} 1 & 0.5 \end{pmatrix} \oplus U_7 \oplus U_8 = \begin{matrix} & 7 & 9 & 10 \\ 7 \\ 9 \\ 10 \end{matrix}\begin{pmatrix} 1.03 & 0.03 & 0 \\ 0.79 & 1.49 & 0.5 \\ 0 & 0.56 & -0.42 \end{pmatrix}$$

Der Knoten 9 ist totally summed und ist bereit für die Elimination. Das Pivot $Piv = 1.49$ sei z.B. akzeptabel. Die Spalte(7) von L: $\begin{matrix} 7 \\ 9 \\ 10 \end{matrix}\begin{pmatrix} 0.0201 \\ 1 \\ 0.3758 \end{pmatrix}$,

$\begin{matrix} & 7 & 9 & 10 \end{matrix}$
die Zeile(7) von U: $\begin{pmatrix} 0.5320 & 1 & 0.3356 \end{pmatrix}$ und zuletzt $\pi(7) = 9$, $\psi(7) = 9$, $D(7) = 1.49$.

Berechne Condition Estimators: $x_+^L = 1 - v_k^L = 1 - (-0.7) = 1.7$, $x_-^L = -1 - v_k^L = -1 - (-0.7) = -0.3$, $x_+^U = 1 - v_k^U = 1 - 1.5 = -0.5$, $x_-^U = -1 - v_k^U = -1 - 1.5 = -2.5$.

$CE_k^L = \max(|x_+^L|, |x_-^L|) = 1.7$, $CE_k^U = \max(|x_+^U|, |x_-^U|) = 2.5$

Es werden alle $\left|\tilde{l}_{j,k}\right| \leq \frac{\varepsilon}{CE_k^L} = \frac{0.9}{1.7} = 0.5294$ und alle $|\tilde{u}_{k,j}| \leq \frac{\varepsilon}{CE_k^U} = \frac{0.9}{2.5} = 0.36$ gedropped.

Wende dropping an: Die Spalte(7) von L: $\begin{matrix} 9 \end{matrix}\begin{pmatrix} 1 \end{pmatrix}$, die Zeile(7) von U: $\begin{matrix} 7 & 9 \end{matrix}$
$\begin{pmatrix} 0.5302 & 1 \end{pmatrix}$, die Update-Matrix: $U_9 = \begin{matrix} 7 \\ 10 \end{matrix}\begin{matrix} \begin{matrix} 7 & 10 \end{matrix} \\ \begin{pmatrix} 1.03 & 0 \\ 0 & -0.42 \end{pmatrix} \end{matrix}$.

Datiere v^L und v^U auf: $Q^L = \emptyset$, $Q^U = \{7\}$,

$\nu_+^U = \|\begin{matrix} 7 \end{matrix}\begin{pmatrix} -0.3 \end{pmatrix} + (-0.5) \cdot \begin{matrix} 7 \end{matrix}\begin{pmatrix} 0.5302 \end{pmatrix}\|_1 = 0.5651$

$\nu_-^U = \|\begin{matrix} 7 \end{matrix}\begin{pmatrix} -0.3 \end{pmatrix} + (-2.5) \cdot \begin{matrix} 7 \end{matrix}\begin{pmatrix} 0.5302 \end{pmatrix}\|_1 = 1.6255$

Daher ist:

$v^L = \begin{matrix} & 3 & 7 & 10 \end{matrix}\begin{pmatrix} 2.3582 & 0.1 & 2.0274 \end{pmatrix}$

$v^U = v^U + x_-^U \tilde{u}_{k,Q^U} = (\ldots) + (-2.5) \cdot \begin{matrix} 7 \end{matrix}\begin{pmatrix} 0.5302 \end{pmatrix} = \begin{matrix} & 3 & 7 & 10 \end{matrix}\begin{pmatrix} 1.7 & -1.6255 & 1.9691 \end{pmatrix}$

Wie bereits erwähnt, ist der Knoten 7 totally summed. Da das Pivot $Piv = 1.03$ z.B. akzeptabel ist, kann man es hier direkt (aus U_9) eliminieren. Die Spalte(8) von L: $\begin{matrix} 7 \end{matrix}\begin{pmatrix} 1 \end{pmatrix}$, die Zeile(8) von U: $\begin{matrix} 7 \end{matrix}\begin{pmatrix} 1 \end{pmatrix}$ und zuletzt $\pi(8) = 7$, $\psi(8) = 7$, $D(8) = 1.03$.

Die Berechnung des Condition Estimators fällt aus, denn es gibt nichts, was gedropped werden könnte. Die Update-Matrix: $U_9 = \begin{matrix} 10 \end{matrix}\begin{matrix} \begin{matrix} 10 \end{matrix} \\ \begin{pmatrix} -0.42 \end{pmatrix} \end{matrix}$.

Da $Q^L = Q^U = \emptyset$, ändert sich bei v^L und v^U nichts.

$v^L = \begin{matrix} & 3 & 10 \end{matrix}\begin{pmatrix} 2.3582 & 2.0274 \end{pmatrix}$, $v^U = \begin{matrix} & 3 & 10 \end{matrix}\begin{pmatrix} 1.7 & 1.9691 \end{pmatrix}$

- Knoten 10: Er besitzt die Knoten 5 und 9 als Söhne.

$$F_{10} = \begin{matrix} & 10 \\ 10 & \end{matrix}\begin{pmatrix} 1 \end{pmatrix} \oplus U_5 \oplus U_9 = \begin{matrix} & \begin{matrix} 3 & 10 \end{matrix} \\ \begin{matrix} 3 \\ 10 \end{matrix} & \begin{pmatrix} 0.44 & 1.3571 \\ -0.37 & 1.1436 \end{pmatrix} \end{matrix}$$

Das Pivot $Piv = 1.1436$ sei z.B. **nicht** akzeptabel. Dessen Elimination wird auf eine spätere Zeitpunkt verschoben (verspätet).

$$U_{10} = F_{10} = \begin{matrix} \\ 3 \\ 10 \end{matrix} \begin{matrix} \begin{matrix} 3 & 10 \end{matrix} \\ \begin{pmatrix} 0.44 & 1.3571 \\ -0.37 & 1.1436 \end{pmatrix} \end{matrix}$$

Eigentlich ist der multifrontale Prozess hier zu Ende und die Matrix U_{10} ist übriggeblieben. Hier kann man bliebige Gauß-Varianten anwenden, z.B. full-Pivoting:

$$\begin{matrix} \\ 3 \\ 10 \end{matrix} \begin{matrix} \begin{matrix} 3 & 10 \end{matrix} \\ \begin{pmatrix} 0.44 & 1.3571 \\ -0.37 & 1.1436 \end{pmatrix} \end{matrix} \xrightarrow[\text{die Spalten}]{\text{permutiere}} \begin{matrix} \\ 3 \\ 10 \end{matrix} \begin{matrix} \begin{matrix} 10 & 3 \end{matrix} \\ \begin{pmatrix} 1.3571 & 0.44 \\ 1.1436 & -0.37 \end{pmatrix} \end{matrix} \xrightarrow{\text{eliminiere}}$$

$$\begin{matrix} \\ 3 \\ 10 \end{matrix} \begin{matrix} \begin{matrix} 10 & 3 \end{matrix} \\ \begin{pmatrix} 1 & 0 \\ 0.8427 & 1 \end{pmatrix} \end{matrix} \begin{pmatrix} 1.3571 & 0 \\ 0 & -0.7408 \end{pmatrix} \begin{matrix} \begin{matrix} 10 & 3 \end{matrix} \\ \begin{pmatrix} 1 & 0.3242 \\ 0 & 1 \end{pmatrix} \end{matrix}$$

Die Spalte(9) von L: $\begin{matrix} 3 \\ 10 \end{matrix} \begin{pmatrix} 1 \\ 0.8427 \end{pmatrix}$, die Zeile(9) von U: $\begin{matrix} 10 & 3 \end{matrix} \begin{pmatrix} 1 & 0.3242 \end{pmatrix}$, $\pi(9) = 3$, $\psi(9) = 10$, $D(9) = 1.3571$ und $\pi(10) = 10$, $\psi(10) = 3$, $D(10) = -0.7408$.

Als Permutationen haben wir also:

$$\pi = (1, 2, 4, 5, 6, 8, 9, 7, 3, 10),\ \psi = (1, 2, 4, 5, 6, 8, 9, 7, 10, 3)$$

und deren Inversen:

$$\pi^{-1} = (1, 2, 9, 3, 4, 5, 8, 6, 7, 10),\ \psi^{-1} = (1, 2, 10, 3, 4, 5, 8, 6, 7, 9)$$

Damit man die endgültige ILU-Zerlegung $A_e(\pi, \psi) \approx LDU$ erhält, müssen wir π^{-1} bzw. ψ^{-1} auf die bisher bestimmten L bzw. U anwenden (grün sind die neuen, endgültigen Indizes):

- Die Spalten von L:

(1): $\begin{matrix} 1 \\ 3 \\ 5 \end{matrix} \begin{pmatrix} 1 \\ 0.7 \\ -0.5 \end{pmatrix} \begin{matrix} 1 \\ 9 \\ 4 \end{matrix}$, (2): $\begin{matrix} 2 \\ 3 \end{matrix} \begin{pmatrix} 1 \\ 0.3 \end{pmatrix} \begin{matrix} 2 \\ 10 \end{matrix}$, (3): $\begin{matrix} 4 \\ 5 \\ 10 \end{matrix} \begin{pmatrix} 1 \\ 0.9 \\ -0.5 \end{pmatrix} \begin{matrix} 3 \\ 4 \\ 10 \end{matrix}$,

(4): $\begin{matrix} 3 \\ 5 \\ 10 \end{matrix} \begin{pmatrix} 0.6909 \\ 1 \\ 0.6364 \end{pmatrix} \begin{matrix} 9 \\ 4 \\ 10 \end{matrix}$, (5): $\begin{matrix} 6 \\ 7 \\ 9 \end{matrix} \begin{pmatrix} 1 \\ 0.1 \\ -0.7 \end{pmatrix} \begin{matrix} 5 \\ 8 \\ 7 \end{matrix}$, (6): $\begin{matrix} 8 \\ 10 \end{matrix} \begin{pmatrix} 1 \\ 0.7 \end{pmatrix} \begin{matrix} 6 \\ 10 \end{matrix}$,

(7): $9 \begin{pmatrix} 0.1 \end{pmatrix} 7$, (8): $7 \begin{pmatrix} 1 \end{pmatrix} 8$, (9): $\begin{matrix} 3 \\ 10 \end{matrix} \begin{pmatrix} 1 \\ 0.8427 \end{pmatrix} \begin{matrix} 9 \\ 10 \end{matrix}$.

- Die Zeilen von U:

(1): $\begin{matrix} 1 & 3 & 5 \\ 1 & 10 & 4 \end{matrix} \begin{pmatrix} 1 & 0.8 & 0.2 \end{pmatrix}$, (2): $\begin{matrix} 2 & 3 & 10 \\ 2 & 10 & 9 \end{matrix} \begin{pmatrix} 1 & 0.9 & 0.2 \end{pmatrix}$, (3): $\begin{matrix} 4 & 10 \\ 3 & 9 \end{matrix} \begin{pmatrix} 1 & -0.7 \end{pmatrix}$, (4): $\begin{matrix} 5 & 10 \\ 4 & 9 \end{matrix} \begin{pmatrix} 1 & -1.3909 \end{pmatrix}$,

$$(5):\begin{pmatrix} 6 & 7 & 9 \\ 1 & -0.3 & 0.7 \\ 5 & 8 & 7 \end{pmatrix},\ (6):\begin{pmatrix} 8 & 9 & 10 \\ 1 & -0.8 & 0.6 \\ 6 & 7 & 9 \end{pmatrix},\ (7):\begin{pmatrix} 7 & 9 \\ 0.5302 & 1 \\ 8 & 7 \end{pmatrix},\ (8):\begin{pmatrix} 7 \\ 1 \\ 8 \end{pmatrix},$$

$$(9):\begin{pmatrix} 10 & 3 \\ 1 & 0.3242 \\ 9 & 10 \end{pmatrix}.$$

Die endgültige ILU-Zerlegung ist $A_e(\pi,\psi) \approx LDU$ mit:

$$A(\pi,\psi) = \begin{pmatrix} 1 & 0 & 0 & 0.2 & 0 & 0 & 0 & 0 & 0 & 0.8 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0.2 & 0.9 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & -0.7 & 0 \\ -0.5 & 0 & -0.9 & 1 & 0 & 0 & 0 & 0 & -0.9 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0.7 & -0.3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & -0.8 & 0 & 0.6 & 0 \\ 0 & 0 & 0 & 0 & -0.7 & 0 & 1 & 1 & 0.5 & 0 \\ 0 & 0 & 0 & 0 & 0.1 & 0 & 0.1 & 1 & 0 & 0 \\ 0.7 & 0 & 0 & 0.9 & 0 & 0 & 0 & 0 & 0.3 & 1 \\ 0 & 0.3 & -0.5 & 0.7 & 0 & 0.7 & 0 & 0 & 1 & -0.1 \end{pmatrix},$$

$$L = \begin{pmatrix} 1 & & & & & & & & & \\ 0 & 1 & & & & & & & & \\ 0 & 0 & 1 & & & & & & & \\ -0.5 & 0 & -0.9 & 1 & & & & & & \\ 0 & 0 & 0 & 0 & 1 & & & & & \\ 0 & 0 & 0 & 0 & 0 & 1 & & & & \\ 0 & 0 & 0 & 0 & -0.7 & 0 & 1 & & & \\ 0 & 0 & 0 & 0 & 0.1 & 0 & 0 & 1 & & \\ 0.7 & 0 & 0 & 0.6909 & 0 & 0 & 0 & 0 & 1 & \\ 0 & 0.3 & -0.5 & 0.6364 & 0 & 0.7 & 0 & 0 & 0.8427 & 1 \end{pmatrix},$$

$$U = \begin{pmatrix} 1 & 0 & 0 & 0.2 & 0 & 0 & 0 & 0 & 0 & 0.8 \\ & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0.2 & 0.9 \\ & & 1 & 0 & 0 & 0 & 0 & 0 & -0.7 & 0 \\ & & & 1 & 0 & 0 & 0 & 0 & -1.3909 & 0 \\ & & & & 1 & 0 & 0.7 & -0.3 & 0 & 0 \\ & & & & & 1 & -0.8 & 0 & 0.6 & 0 \\ & & & & & & 1 & 0.5302 & 0 & 0 \\ & & & & & & & 1 & 0 & 0 \\ & & & & & & & & 1 & 0.3242 \\ & & & & & & & & & 1 \end{pmatrix}$$

und $D = \mathrm{diag}(1, 1, 1, 1.1, 1, 1, 1.49, 1.03, 1.3571, -0.7408)$.

Die Restmatrix $R = A_e(\pi, \psi) - LDU$

$$= \begin{pmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1.0\,\mathrm{e}{-5} & 4.0\,\mathrm{e}{-1} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 2.0\,\mathrm{e}{-6} & 5.0\,\mathrm{e}{-1} & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 3.0\,\mathrm{e}{-2} & 0 & 0 & 0 \\
0 & 0 & 0 & 1.0\,\mathrm{e}{-5} & 0 & 0 & 0 & 0 & -3.0\,\mathrm{e}{-5} & 2.8\,\mathrm{e}{-5} \\
0 & 0 & 0 & -4.0\,\mathrm{e}{-5} & 0 & 0 & 5.6\,\mathrm{e}{-1} & 5.7\,\mathrm{e}{-5} & 5.7\,\mathrm{e}{-5} & 3.6\,\mathrm{e}{-5}
\end{pmatrix}$$

und der inverse Fehler $L^{-1}RU^{-1}D^{-1} = L^{-1}A_e(\pi, \psi)U^{-1}D^{-1} - I$

$$= \begin{pmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -7.4\,\mathrm{e}{-6} & -5.4\,\mathrm{e}{-1} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1.9\,\mathrm{e}{-6} & 3.7\,\mathrm{e}{-1} & 2.2\,\mathrm{e}{-1} \\
0 & 0 & 0 & 0 & 0 & 0 & 2.0\,\mathrm{e}{-2} & -1.5\,\mathrm{e}{-2} & 0 & 0 \\
0 & 0 & 0 & 9.1\,\mathrm{e}{-6} & 0 & 0 & 0 & 0 & -6.7\,\mathrm{e}{-6} & 3.7\,\mathrm{e}{-1} \\
0 & 0 & 0 & -4.4\,\mathrm{e}{-5} & 0 & 0 & 3.7\,\mathrm{e}{-1} & -2.9\,\mathrm{e}{-1} & 1.2\,\mathrm{e}{-5} & 2.9\,\mathrm{e}{-2}
\end{pmatrix}$$

vermitteln ein Gefühl für die Güte des Präkonditionierers.

7.3 Wie das Programm vorgeht

7.3.1 Input erzeugen

Das Programm braucht erstens eine Datei, in der die (Input-) Matrix A_{inp} gespeichert ist. Meine Testmatrizen stammen aus den Web-Repositorien "MatrixMarket"[9] und "University of Florida Sparse Matrix Collection"[10]. Die Matrizen stehen dort als MATLAB ".mat"-Files zur Verfügung. Da ich keine ".mat"-files direkt lesen konnte, musste ich Konvertierungsroutinen schreiben, welche das Input für mein Programm erzeugen. Ich speichere die Matrizen im (ganz normalen) Koordinatenformat:

[9] http://math.nist.gov/MatrixMarket/
[10] http://www.cise.ufl.edu/research/sparse/matrices/

Z.B. sieht für die (sparse) Matrix $A = \begin{pmatrix} 1.1 & 0 & 0 \\ 0 & 0 & 2.3 \\ 0 & 0 & 0 \end{pmatrix}$ ein mögliches "input_file.txt" so aus:

$$\begin{array}{ll} \text{n} & \\ \text{nnz} & \\ \text{rn} & \\ \text{cn} & \\ \text{val} & \end{array}$$

3	$\longrightarrow n$
2	$\longrightarrow nnz$
1	$\left.\begin{matrix}\\\\\end{matrix}\right\}$ rn
2	
1	$\left.\begin{matrix}\\\\\end{matrix}\right\}$ cn
3	
1.1	$\left.\begin{matrix}\\\\\end{matrix}\right\}$ val
2.3	

Das Programm ist für unsymmetrische Matrizen gedacht. Daher ist es nicht vorgesehen nur Teile der Matrizen zu speichern.

Nachdem die Matrix gelesen wird, geht es mit der Preprocessing-Phase weiter.

7.3.2 Preprocessing 1

Extrahiere die dominante Komponente: $A_{inp} \longrightarrow A$

Am Anfang dieses Kapitels wurde erklärt, wieso man für die multifrontale Methode beidseitig irreduzible Matrizen braucht. Dafür bestimme ich eine BRNF für A_{inp}:

$$P_{inp} A_{inp} Q_{inp}^T = \begin{pmatrix} A_{1,1} & & 0 \\ & \ddots & \\ * & & A_{N,N} \end{pmatrix}$$

wobei alle Diagonalblöcke $A_{i,i}$ nullfreie Diagonalen besitzten und alle $G(A_{i,i})$ stark zusammenhängend sind. Unter diesen Diagonalblöcken wähle ich denjenigen mit maximaler Dimension:

$$A = A_{j,j}, \text{ wobei } \dim(A_{j,j}) \geq \dim(A_{i,i}) \ \forall i = 1, \ldots, N$$

Ab diesem Zeitpunkt, mache ich mit A weiter. Sei $n = \dim(A)$.

7.3.3 Preprocessing 2

Transformation auf eine I-Matrix

Es werden zwei Vektoren u, v und eine Permutation[11] η (alle der Länge n) bestimmt, sodass die Matrix $(\widehat{U}A\widehat{V})P_\eta^T$ eine I-Matrix ist. Hier ist $\widehat{U} = \mathrm{diag}(u)$, $\widehat{V} = \mathrm{diag}(v)$ und P_η die Permutationsmatrix, die η entspricht.

Die Transformation auf eine I-Matrix $(\widehat{U}A\widehat{V})P_\eta^T$ bringt eine Verbesserung der Konditionszahl. Dadurch werden die numerischen Ergebnisse vertrauenswürdiger.

Andererseits habe ich beobachtet, dass wenn $\widehat{U}$ oder $\widehat{V}$ riesige Einträge enthalten, die Matrix A schlecht konditioniert war. Das hat einige Folgen, welche beim GMRES Schritt besprochen werden.

Die Algorithmen hierfür sind im Kapitel 2 detalliert beschrieben worden. Da meine eigene Implementierung langsamer war als die von Duff und Koster, habe ich deren Routine MC64 in meinen Code eingebunden.

7.3.4 Preprocessing 3

Bestimme eine symmetrische MinDeg Permutation und wende sie an

Hier wird eine MinDeg Permutation bezüglich (der symmetrischen Hülle von) $(\widehat{U}A\widehat{V})P_\eta^T$ bestimmt. Dieser Schritt liefert eine Permutation μ, sodass bei einer (pivotfreien) LU-Zerlegung von

$$P_\mu((\widehat{U}A\widehat{V})P_\eta^T)P_\mu^T$$

wesentlich weniger Fill-in entsteht als bei einer (pivotfreien) LU-Zerlegung von $(\widehat{U}A\widehat{V})P_\eta^T$. Die neue Matrix $P_\mu((\widehat{U}A\widehat{V})P_\eta^T)P_\mu^T$ ist immer noch eine I-Matrix.

Der verwendete Algorithmus ist der **amd** (Approximate Minimum Degree) Algorithmus von Amestoi, Davis und Duff. Dieser Algorithmus wurde im Kapitel 3 beschrieben. Der Code ist frei erhältlich unter http://www.cise.ufl.edu/research/sparse/amd/ (die Seite wird von Timothy Davis gewartet).

7.3.5 Preprocessing 4

Bestimme den Eliminationsbaum, ein Postordering und wende sie an

Mit der (zusammenhängenden) symmetrischen Hülle der Matrix

$$P_\mu((\widehat{U}A\widehat{V})P_\eta^T)P_\mu^T$$

als Ausgangspunkt wird der Eliminationsbaum bestimmt, in Gestalt eines Vektors $Parent$[12] (der Länge n). Der dazu verwendeter Algorithmus wurde eingehend beschrieben und formuliert auf Seite 73.
Um aus dem Vektor $Parent$ ein Postordering e zu berechnen, muss man den

[11] eigentlich eine Spaltentransversale
[12] Um klar zu machen, dass n eine Wurzel ist, setzt man $Parent(n) = 0$

Eliminationsbaum in ein kompaktähnlichen Format $(sst, snr, sons)$ konvertieren: $sst \rightarrow$ sons start, $snr \rightarrow$ sons number, $sons \rightarrow$ sons. Der Vektor $sons(sst(i) : sst(i) + snr(i) - 1)$ soll uns all die Söhne eines Knoten i zur Verfügung stellen. Es genügt, wenn die Länge des Vektors $sons$ $(n-1)$ ist, weil abgesehen von der Wurzel alle anderen Knoten auch Söhne sind. Folgender Algorithmus schafft das in $\mathcal{O}(n)$:

Algorithmus 7.2 Bestimme ein kompaktähnliches Format $(sst, snr, sons)$

Beschreibung: Input: $Parent$; Output $(sst, snr, sons)$

1: $sst = snr = zeros(1, n)$, $sons = zeros(1, n-1)$, $sst(1) = 1$
2: **for** $i = 1 : n-1$ **do** {zähle Söhne aller Parens}
3: $\quad snr(Parent(i)) + +$
4: **end for**
5: **for** $i = 2 : n$ **do** {initialisiere sst}
6: $\quad sst(i) = sst(i-1) + snr(i-1)$
7: **end for**
8: $tmp = sst$
9: **for** $i = 1 : n-1$ **do** {trage Söhne ein}
10: $\quad j = Parent(i)$, $sons(tmp(j)) = i$, $tmp(j) + +$
11: **end for**

Mit dem gerade errechneten kompaktähnlichen Format $(sst, snr, sons)$ als Eingabe, bestimmt der folgende Algorithmus in $\mathcal{O}(n)$ eine Postordering. Wegen der Effizienz habe ich ihn ohne Rekursion implementiert:

Algorithmus 7.3 Bestimme ein Postordering e

Beschreibung: Input: $(sst, snr, sons)$; Output e

1: $e = stack = tmp = zeros(1, n)$, $e\_count = stack\_size = 0$
2: $stack\_size = 1$, $stack(1) = n$ {füge n in Stack ein}
3: **while** $stack\_size > 0$ **do** {Stack nicht leer}
4: $\quad u = stack(stack\_size)$
5: $\quad$ **if** $tmp(u) < snr(u)$ **then** {u hat noch nicht besuchte Söhne}
6: $\quad\quad v = sons(sst(u) + tmp(u))$
7: $\quad\quad tmp(u) + +$
8: $\quad\quad (stack\_size + +)$, $stack(stack\_size) = v$ {füge v in Stack ein}
9: $\quad$ **else** {u ist fertig}
10: $\quad\quad (stack\_size - -)$ {entferne Element aus dem Stack}
11: $\quad\quad (e\_count + +)$, $e(e\_count) = u$
12: $\quad$ **end if**
13: **end while**

Wende Postordering an:

$$P_\mu((\widehat{U}A\widehat{V})P_\eta^T)P_\mu^T \longrightarrow P_e(P_\mu((\widehat{U}A\widehat{V})P_\eta^T)P_\mu^T)P_e^T = \widetilde{A}$$

Man darf nicht vergessen, diese Permutation e auch auf der Eliminationsbaum anzuwenden. Dies geschiet nach den folgenden Regeln:

$$Parent(1:n-1) = e^{-1}(Parent(1:n-1))$$

$$sst = sst(e)$$

$$snr = snr(e)$$

$$sons = e^{-1}(sons)$$

7.3.6 Hauptroutine: Berechnung der ILU

Basierend auf der multifrontalen Methode mit verzögerter Pivotisierung bestimmt die Hauptroutine eine unvollständige LU Zerlegung mit Pivotisierung. Sie liefert zwei Permutationen π, ψ der Länge n, eine $(n \times n)$ sparse untere Dreiecksmatrix L mit Einheitsdiagonale[13], eine $(n \times n)$ Diagonalmatrix[14] D und eine $(n \times n)$ sparse obere Dreiecksmatrix U ebenfalls mit Einheitsdiagonale, sodass:

$$C = P_\pi \widetilde{A} P_\psi^T \approx LDU. \tag{7.1}$$

Diese Routine liefert auch Informationen über das Fill-in. Basierend auf einem Algorithmus (siehe Seite 74), welcher aus dem Eliminationsbaum die Fill-in Informationen errechnet, wird eine Schranke für die Größe des Fill-ins mitgeliefert. Dieses Algorithmus berücksichtigt keinerlei Pivotisierung. Aber für die Beurteilung des Fill-in ist er ganz gut zu gebrauchen. Die verzögerte Pivotisierung zieht von Natur aus sehr wenig Fill-in nach sich (siehe Satz 8.1). Dadurch wird die bereits angewandte Minimum Degree Permutation nur minimal "gestört".

Es folgt eine kurze Diskussion über die Pivotisierungsstrategie. Da ich es mit I-Matrizen zu tun habe, entschied ich mich für folgende **Pivotisierungsstrategie**:

"Akzteptiere Piv als Pivot, falls $|Piv| \geq piv\_tol$, wobei piv_tol vorgegeben ist"

I-Matrizen legen für piv_tol Werte aus $(0, 1)$ nahe. Diese Strategie schien mir sinnvoll für I-Matrizen. Das Kriterium ist sofort überprüfbar und braucht genau einen Vergleich. In Kombination mit der verzögerten Pivotisierung ist das, meiner Meinung nach, eine gute Herangehensweise. Als Startwert benutze ich ausschließlich $piv\_tol = 0.5$. Oft ist ein fine tuning nötig, um einen optimalen Wert für

[13] redundante Diagonalen werden nicht explizit gespeichert

[14] genauer gesagt, nur deren Diagonale

piv_tol zu bestimmen. Leider gibt es keinen allgemeingültigen optimalen Wert für *piv_tol*. Solches Parameter fine tuning ist bei ILU Methoden immer nötig. Das kommt daher, dass die Größe des Fill-in nie a priori geschätzt werden kann.

Unterwegs versuche ich, verspätete Knoten zu eliminieren, und zwar so: Nach jeder geglückten Elimination versuche **höchstens einen** (wie z.B. in [45] vorgeschlagen) verspäteten Knoten aus der gleichen frontale Matrix (sonst macht es kein Sinn) zu eliminieren.

Verzögerte Pivotisierung hat ihre Grenzen. Bleiben zu viele Knoten verspätet, so hat das Folgen für das Fill-in. Zwar bringt ein verspäteter Knoten sehr moderaten extra Fill-in. Je länger ein Knoten verspätet bleibt und je mehr Knoten verspätet werden, desto größer wird das zusätzliche Fill-in werden. Daher überprüfe ich in jedem Schritt, wie viele verspätete Knoten es gibt. Wird eine (durch die Eingabe mitgelieferte) Schwellenwertvariable *delayed_nodes_upperbound* überschritten, so bricht das Programm ab mit einen failure Meldung. Tritt dieser Fall auf, sollte man fine tuning betreiben, z.B *piv_tol* nach unten korrigieren. Ein akzeptabler Wert für *delayed_nodes_upperbound* ist 300.

Last but not least, braucht die ILU Routine einen Wert ε (als Input) für das Dropping durch den condition estimator. In meinen Programmen benutze ich statt ε den Namen τ. Anfangswerte $\tau \in [0.1, 0.5]$ sind aus Erfahrung akzeptabel. Bezüglich des condition estimator dürfte aus dem oben detailliert beschriebenen Beispiel (Abschnitt 7.2) alles klar sein. Wie immer ist ein fine tuning nicht ausgeschlossen.

7.3.7 Lösung des präkonditionierten System mittels GMRES

Es wird ein lineares Gleichungssystem $Ax = b$ generiert, wobei $x, b \in \mathbb{R}^n$, mit b zufällig. Wie immer ist $r_x = b - Ax$. Das System kann äquivalent wie folgt umgewandelt werden:

$$Ax = b$$

$$\Updownarrow$$

$$\underbrace{(P_\pi P_e P_\mu \widehat{U} A \widehat{V} P_\eta^T P_\mu^T P_e^T P_\psi^T)}_{C} \underbrace{(P_\psi P_e P_\mu P_\eta \widehat{V}^{-1} x)}_{y} = \underbrace{(P_\pi P_e P_\mu \widehat{U} b)}_{d} \tag{7.2}$$

Die Matrizen C und d werden in meinem Programm explizit berechnet. Hat man die Lösung y des Systems $Cy = d$, so kann die Lösung x des Anfangssystem aus:

$$x = \widehat{V} P_\eta^T P_\mu^T P_e^T P_\psi^T y \tag{7.3}$$

bestimmt werden. Es gilt: $r_y = d - Cy = P_\pi P_e P_\mu \widehat{U} r_x$.

Es folgt der Aufruf des split-preconditioned GMRES(m) bezüglich des LGS $Cy = d$. Das ist im Grunde ein ganz normales GMRES(m) bezüglich des LGS:

$$\underbrace{(L^{-1}CU^{-1}D^{-1})}_{T}\underbrace{(DUy)}_{z} = \underbrace{(L^{-1}d)}_{h} \tag{7.4}$$

Die Matrizen T und h werden nie explizit berechnet (siehe nachfolgender Algorithmus). Es gilt: $r_z = h - Tz = L^{-1}r_y$.

Um GMRES zu beenden, brauchen wir ein Abbruchkriterium. Die Residuen müssen dabei berücksichtigt werden. GMRES löst (annähernd) das LGS $Cy = d$ und nicht $Ax = b$. Es ist oft der Fall, dass die Normen der jeweiligen Residuen weit weg voneinander liegen. Es ist also nötig, parallel zu $\|r_y\| = \|d - Cy\|$, auch das Residuum $\|r_x\| = \|b - Ax\|$ in jedem Schritt zu bestimmen.

Aus (7.3) folgt:

$$r_x = \widehat{U}^{-1}P_\mu^T P_{\mathrm{e}}^T P_\pi^T r_y$$

Diese Formel ist eher umständlich, die Matrix $\widehat{U}^{-1}P_\mu^T P_{\mathrm{e}}^T P_\pi^T$ ist z.B. nicht diagonal. Es gibt aber eine einfach zu berechnende Diagonalmatrix $\widehat{D}$, sodass[15]:

$$\|r_x\|_p = \|\widehat{D}\, r_y\|_p.$$

Also reduziert sich die Berechnung von $\|r_x\|_p$ auf nur eine Multiplikation $Diagonalmatrix \times Vektor$.

Die Vektornorm $\|\cdot\|_p$ ist invariant gegenüber Permutationen[16]. Aus:

$$P_\mu^T P_e^T P_\pi^T = (P_\pi P_e P_\mu)^T = P_{\mu\circ e\circ\pi}^T$$

und

$$r_x = P_{\mu\circ e\circ\pi}^T (P_{\mu\circ e\circ\pi}\widehat{U}^{-1}P_{\mu\circ e\circ\pi}^T) r_y$$

folgt

$$\|r_x\|_p = \|P_{\mu\circ e\circ\pi}^T \widehat{D}\, r_y\|_p = \|\widehat{D}\, r_y\|_p$$

mit

$$\widehat{D} = P_{\mu\circ e\circ\pi}\widehat{U}^{-1}P_{\mu\circ e\circ\pi}^T$$

Offensichtlich ist $\widehat{D}$ eine Diagonalmatrix. Um sie zu berechnen, braucht man nur die Permutationenverkettung $(\mu \circ e \circ \pi)$ auf die Diagonale von $\widehat{U}^{-1}$ anzuwenden. Am besten ist es, $\widehat{D}$ als Parameter an die GMRES zu übergeben (siehe folgender Algorithmus).

[15] $1 \le p \le +\infty$, wie üblich

[16] Eine andere Begründung für den Fall $p = 2$: Permutationsmatrizen sind orthonormal

Algorithmus 7.4 Repeated split-preconditioned GMRES(m)

Beschreibung: Input: $C, d, L, D, U, \widehat{D}, y_0, m, step\_nr, thresh\_res$; Output y

1: $V = zeros(n, m+1)$, $H = zeros(m+1, m)$, $y = y_0$
2: **for** $step = 1; step \leq step\_nr; step++$ **do**
3: $r_y = d - Cy$
4: **if** $\|\widehat{D} r_y\|_2 < thresh\_res$ **then**
5: **return** y {Zielresiduum erreicht: $\|r_x\|_2 < thresh\_res$ }
6: **end if**
7: $V(:,1) = L^{-1} r_y$ {$L^{-1} r_y$ entspricht eigentlich r_z}
8: $\gamma = \|V(:,1)\|_2$, $V(:,1) = V(:,1)/\gamma$
9: **for** $j = 1 : m$ **do**
10: $V(:, j+1) = L^{-1} C U^{-1} D^{-1} V(:,j)$ {sparse Operationen}
11: **for** $i = 1 : j$ **do** {orthogonalisiere $V(:, j+1)$ gegenüber $V(:, 1:j)$}
12: $H(i,j) = \langle V(:,j+1), V(:,i) \rangle$
13: $V(:, j+1) = V(:, j+1) - H(i,j) V(:,i)$
14: **end for**
15: $H(j+1, j) = \|V(:, j+1)\|_2$, $V(:, j+1) = V(:, j+1)/H(j+1, j)$
16: **end for**
17: finde $w \in \mathbb{R}^m$ sodass $\|\underbrace{(\gamma, 0, \ldots, 0)}_{m+1}^T - Hw\|_2$ minimal {QR, m Givens Rot.}
18: $y = y + U^{-1} D^{-1} V(:, 1:m) w$
19: **end for**
20: $r_y = d - Cy$
21: **if** $\|\widehat{D} r_y\|_2 < thresh\_res$ **then**
22: **return** y {Zielresiduum erreicht: $\|r_x\|_2 < thresh\_res$ }
23: **end if**
24: **return** Zielresiduum nicht erreicht

Die Parametern $step\_nr, thresh\_res$ sind ähnlich wie bei Bollhöfer[17] ausgewählt (es ist übrigens eine Standardauswahl). GMRES(m) kann man als eine Methode mit m Minischritten betrachten. Üblich ist GMRES(30) mit höchstens 500 Minischritten, d.h. höchstens $\frac{500}{30} \approx 17$ (Groß) Schritten GMRES(30). Daher verwenden wir $step\_nr = 17$. Bollhöfers GMRES bricht erfolgreich ab, wenn $\|r_x\| \leq \sqrt{\epsilon_{\text{machine}}} \|b - Ax_0\| = 2^{-26} \|b - Ax_0\| \approx 10^{-8} \|r_{x_0}\|$. So gehe ich auch vor: $thresh\_res = 10^{-8} \|r_{x_0}\|$.

Manchmal tritt folgende merkwürdige Situation auf: Während $\|r_y\|$ sehr klein ist, ist $\|r_x\|$ groß (z.B. um einen Faktor 10^6 größer). Das kann passieren, wenn $\widehat{U}$ oder $\widehat{V}$ (siehe Schritt "Transformation auf I-Matrix") einige riesige Einträge enthalten. Das ist ein (heuristisches) Indiz dafür, dass die Matrix A schlecht konditioniert ist.

[17] Bollhöfer bindet **sparskit** ([56]) von Saad ein, dessen Bestandteil GMRES ist.

Ein großes Manko aller iterativer Methoden ist, dass ihr Abbruch auf einem (sehr) kleinen Residuum beruht, aber keine Abschätzung für die Konditionszahl der Matrix A bestimmt werden kann. Hierzu zählt auch meine Methode. Wenn die Konditionszahl von A riesig ist, ein Vektor $\tilde{x}$ mit $\|r_{\tilde{x}}\|$ sehr klein **muss nicht** zwangsläufig auch nahe an der exakte Lösung x^* liegen, wie das folgende Beispiel zeigt:

$n = 12$, $A = (\frac{1}{i+j-1})_{i,j=1,\ldots,n}$ (Hilbert Matrix), $x^* = ones(n, 1)$, $b = Ax^*$ und $\tilde{x} = (1, 1, 0.9999, 1.0009, 0.9931, 1.0302, 0.9162, 1.1512, 0.8232, 1.1292, 0.9464, 1.0096)$.

Obwohl das Residuum $\|r_{\tilde{x}}\|_\infty = \|b - A\tilde{x}\|_\infty = 8.8818e - 16$ extrem kein ist, beträgt der relative Fehler:

$$\frac{\|\tilde{x} - x^*\|_\infty}{\|x^*\|_\infty} = 17.68\%$$

was $\tilde{x}$ zu einer schlechten Näherung für die exakte Lösung x^* macht.

Für eine effiziente Implementierung habe ich in meiner GMRES selbstverständlich auch BLAS (Zeile 3,4,8,12,13,15,18,20,21) und Sparse-BLAS ([1]) (Zeilen 3,7,10,18,20) Routinen verwendet.

Kapitel 8

Implementierungsaspekte und eine zweite Implementierung

8.1 Implementierungsaspekte

Bei der Umwandlung einer direkten multifrontalen Methode zu einer ILU, gibt es mehrere Aspekte, die anders implementiert werden sollten. Bei der direkten multifrontalen Methode entsteht der Aufwand an zwei Stellen des Algorithmus (siehe Seite 83):

1. die Assembly-Phase:

$$F_j = \begin{pmatrix} A_{j,j} & A_{j,\mathcal{L}_j} \\ A_{\mathcal{L}_j,j} & 0 \end{pmatrix} \oplus U_{c_1} \oplus \ldots \oplus U_{c_s}$$

2. die Eliminationsphase:

$$F_j = \begin{pmatrix} 1 & 0 \\ L_{\mathcal{L}_j,j} & I \end{pmatrix} \begin{pmatrix} D_j & 0 \\ 0 & U_j \end{pmatrix} \begin{pmatrix} 1 & U_{j,\mathcal{L}_j} \\ 0 & I \end{pmatrix}$$

Nachdem ich das dropping eingebaut hatte, beobachtete ich, dass naiv implementiert (mit 2-dimensionalen Arrays) die Assembly-Phase über 90% der ILU-Laufzeit in Anspruch nahm. Andererseits habe ich auch beobachtet, dass die Update-Matrizen U_{c_j} sparse waren. Je besser dropping wirkte, desto sparser wurden sie. Somit beschloss ich, diese Matrizen als sparse zu implementieren.

Bei der Assembly-Phase braucht man Zugriff auf alle Nicht-Null-Einträge der Update-Matrizen. Das bedeutet, dass bei einer sparse Implementierung das verwendete Format es uns leicht machen sollte, die Nichtnullen zu generieren und auf

sie zuzugreifen. Formate, die die Matrix zeilenweise (oder spaltenweise) speichern, sind einfach zu handhaben und sinnvoll für unser Problem.

Die erste Idee ist, die Zeileneinträge in Listen zu speichern. Listenformate sind von Natur aus sehr flexibel. Das stellt sich für unser Problem aber schnell als ungeeignet heraus. Mit einer Liste der Länge m kann man zwar die Zeileneinträge in $\mathcal{O}(m)$ leicht generieren, aber auf eine bestimmte Koordinate zuzugreifen oder zu überprüfen, ob ein bestimmter Eintrag existiert (nötig bei der erweiterten Addition $\oplus$), ist aufwendig (es bedarf eines $\mathcal{O}(m)$-Durchlaufs bis das gesuchte Element auftritt). Solche Probleme werden auch bei der Eliminationsphase auftreten.

Es gibt Datenstrukturen, die für solche Operationen besser geeignet sind, z.B. AVL-Bäume ([52]). In einen AVL-Baum mit m Knoten, benötigen alle Grundoperationen *Suchen, Einfügen, Entfernen* eine Aufwand von $\mathcal{O}(\log m)$. Es ist außerdem möglich, die Datensätze (die in den AVL-Knoten gespeichert sind) in $\mathcal{O}(m)$ zu sortieren. Aus diesen Gründen entschied ich, die Zeilen der update Matrizen als AVL-Bäume zu implementieren. Dadurch ist eine update Matrix bei mir ein Array $avl\_baum * rows$, wobei jeder Eintrag $rows(j)$ ein AVL-Baum ist. Folgende Abbildung verdeutlicht dies:

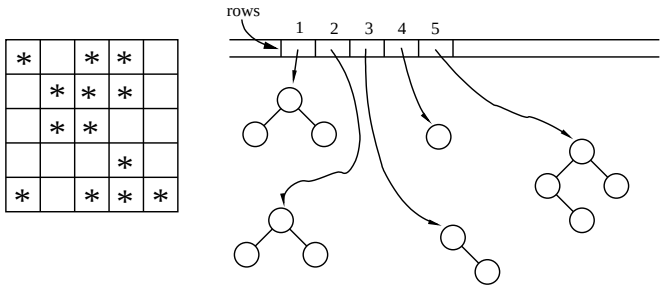


Abbildung 8.1:

Ich möchte etwas genauer beschreiben, wie mein Sparse-Format aussieht. Für ein umfassendes Verständnis, müssen auch die AVL-Knoten und AVL-Bäume kurz beschrieben werden.

Die Preprocessing-Routinen habe ich zuerst in MATLAB implementiert, und ich wollte die Indizierung "ab 1" beibehalten. Das kann wirklich zum Problem werden bei einer "Übersetzung" in die Sprache C. Es gibt einige Möglichkeiten, dieses Problem zu lösen. Eine Möglichkeit wäre, jedes Array um 1 länger anzulegen und den Index 0 zu vermeiden. Eine andere Möglichkeit (abgesehen von der expliziten Subtraktion von 1[1]) ist, ein Makro mit zwei Parametern zu definieren, wie folgt:

```
#define aks(array,index) ((array)[(index)-1])
```

[1] macht den Code unlesbar

Eine dritte Möglichkeit ist eine Wrapper-Klasse um jedes Array herum zu legen und den Indizierungsoperator `[]` zu überladen:

```
template<class T>
struct my_Array
{
   int size;
   T *data;
   my_Array(int n):size(n)
   {   data=new T[n];}
   T& operator[](int index)
   {   return data[index-1];}
   ~my_Array()
   {   delete[] data;}
}
```

Ich habe mich für die Makro-Variante entschieden, die anderen scheinen mir zu umständlich. Im Folgenden, sind die Datentypen `myint` und `key_typ` so definiert:

```
typedef unsigned int myint;
typedef myint key_typ;
```

In meinem ganzen Programm habe ich negative ganzzählige Variablen nicht gebraucht. Daher wollte ich den in C darstellbaren Bereich nicht unnötig einschränken. Jede frontale Matrix enthält zwei Indextypen, **lokale** und **globale**. Als Beispiel sei die frontale Matrix (mit einem verspäteten Knoten)

$$F = \begin{array}{c} \\ 6 \\ 7 \\ 9 \end{array} \begin{array}{c} \begin{array}{ccc} 6 & 7 & 9 \end{array} \\ \begin{pmatrix} 1.0 & -0.3 & 0.7 \\ 0.1 & 2.3 & 0 \\ -0.7 & 0 & 3.4 \end{pmatrix} \end{array}$$

gegeben. Dies ist eine 3×3 Matrix. Die Indizes 1,2,3 für die Zeilen und Spalten diese Matrix habe ich lokal genannt, im Unterschied zu 6,7,9, die ich global genannt habe. Die frontalen Matrizen sind nur "Ausschnitte" einer großen globalen Schur-Update-Matrix.

Jeder Eintrag einer frontalen Matrix wird als Knoten in einen AVL-Baum gespeichert. Abgesehen von Informationen für die eigene Verwaltung der Knoten im AVL-Baum, habe ich die Eintragsinformationen in einer Struktur `datensatz` gespeichert:

```
struct datensatz
{
   key_typ col_ind; //global Index, dient gleichzeitig auch als
Knotenschlüssel
   key_typ row_ind; //global Index
   double val;
};
```

Beachte, dass `col_ind` und `row_ind` globale Indizes sind. Z.B. bei der Matrix F, für den Eintrag $F_{2,1} = 0.1$ wären dies: $col\_ind = 6$, $row\_ind = 7$ und $val = 0.1$. Da ich die Knoten zeilenweise in AVL-Bäumen speichern wollte, habe ich entsprechend den Strukturmember `col_ind` als **Knotenschlüssel** benutzt. Wieso werden nicht die lokalen Indizes, statt der globalen, gespeichert? Die globalen Indizes haben einen entscheidenden Vorteil. Sie erleichtern uns sehr den Assembly-Prozess (wird unten genauer erläutert).

Um die Einträge in einen AVL-Baum zu integrieren, habe ich die Datentypen `avl_knoten` und `avl_baum` wie folgt definiert:

```
struct avl_knoten
{
   datensatz knoten_data;
   int bal; // balance: Werte aus {-1,0,+1}
   struct avl_knoten *parent;
   struct avl_knoten *lsohn;
   struct avl_knoten *rsohn;
};
typedef avl_knoten *avl_baum;
```

Es dürfte klar sein, was die Members `bal`, `parent`, `lsohn` und `rsohn` repräsëntieren. Als Beispiel steht unten die Zeile 1 von F im oben beschriebenen Format. Beachte die Knotenschlüssel `col_ind`, welche grün dargestellt werden.

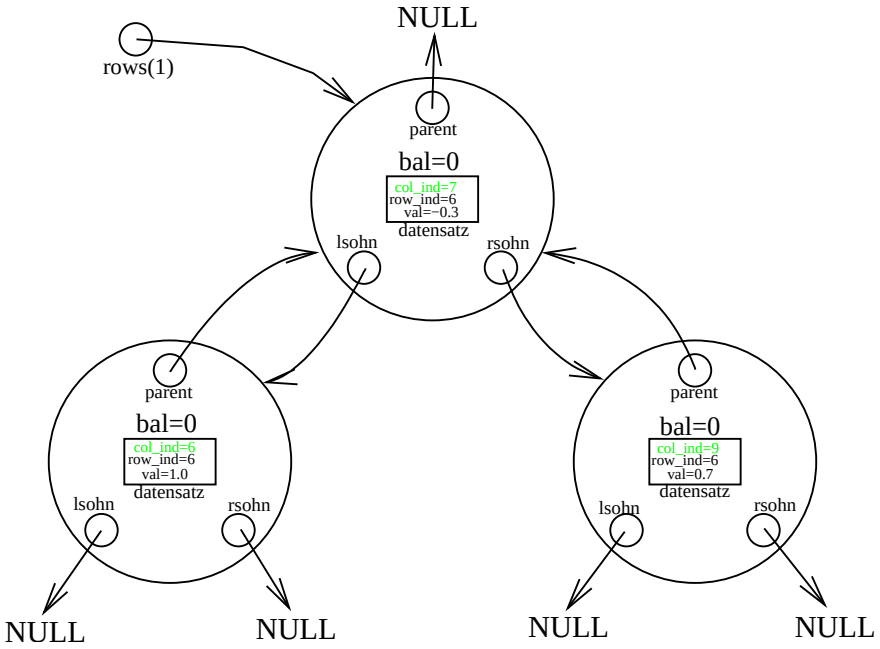


Abbildung 8.2:

Damit das Zusammenspiel mit der Gauß-Elimination funktioniert, braucht der Datentyp `avl_baum` einige Methoden, die unten aufgelistet sind:

```
namespace avl_operations
{
   avl_knoten* avl_suchen(key_typ k, avl_baum b);
   void avl_einfuegen(datensatz d, avl_baum *bp);
   void avl_loeschen(avl_knoten *z, avl_baum *bp);
   void avl_baum2array(avl_baum b, avl_knoten **knoten_array, myint
*ptr2size);
   void avl_deallocate(avl_baum *bp);
}
```

Von der Effizienz dieser Grundroutinen hängt in große Maße die Performance des Programms ab. Die drei ersten Routinen habe ich aus Effizienzgründen ganz ohne Rekursionen implementiert. Was diese Routinen machen, brauche ich nicht zu erklären. Die vierte Routine generiert eine Zeile als 1-dimensionale Array anhand ihrer AVL-Baum-Darstellung. Die fünfte Routine löscht eine (als AVL-Baum dargestellte) Zeile und gibt den verwendeten Speicher wieder frei. Diese zwei Routinen wurden rekursiv implementiert.

Um Arrays anzulegen und gleichzeitig mit Nullen zu initialisieren, benutze ich folgende Routinen:

```
inline myint* zeros(myint n)
{
   return (myint *)calloc( n , sizeof(myint) );
}

template <typename T>
inline void zeros(T **ptr, myint n)
{
   *ptr = (T *)calloc( n ,sizeof(T));
}
```

Als nächstes wird der Datentyp `ext_sqr_matr` vorgestellt. Dieser implementiert die frontalen Matrizen.

- m ist die Dimension der frontalen Matrix. Bei unserer Beispielmatrix F ist $m = 3$.

- dd ist die Anzahl der verspäteten Knoten, die die frontale Matrix enthält. Bei F aus dem Beispiel ist $dd = 1$.

- nnz ist die Anzahl der Nichtnullen, die die frontale Matrix enthält. Im Beispiel ist $nnz = 7$.

Der Datentyp für die frontalen Matrizen:

```
nodes_inv=new myint[n]; // n ist die Dimension des Problems
struct ext_sqr_matr
{
   myint m;
   myint dd;
   myint nnz;
   myint *nodes;
   avl_baum *rows;

   ext_sqr_matr(myint m_inp, myint dd_inp=0, myint nnz_inp=0) :
m(m_inp), dd(dd_inp), nnz(nnz_inp)
   { nodes=zeros(m); zeros(&rows,m); }

   avl_knoten* operator () (myint i, myint jj)
   { return avl_operations::avl_suchen(jj,aks(rows,i)); }

   avl_knoten* operator () (myint k)
   { return avl_operations::avl_suchen(aks(nodes,k),aks(rows,k)); }

   void neue_element_einfuegen(myint i, myint jj, double val)
   {
      datensatz d; d.row_ind=aks(nodes,i); d.col_ind=jj; d.val=val;
      avl_operations::avl_einfuegen( d , rows+i-1 );
      nnz++;
   }

   void invert_node_array()
   {
      for(myint j=1; j<=m; j++)
         aks(nodes_inv,aks(nodes,j))=j;
   }

   ~ext_sqr_matr()
   {
      for(myint k=0; k<m; k++)
         avl_operations::avl_deallocate(rows+k);
      free(rows);
      free(nodes);
   }

};
```

- *nodes* ist ein Array der Länge $\geq m$, welches die globalen Indizes der frontalen Matrix in aufsteigender Reihenfolge speichert. Bei F ist $nodes = [6, 7, 9]$.

- *rows* ist ein Array der Länge $\geq m$, welches die Zeilen der frontalen Matrix als AVL-Bäume speichert. Abbildungen 8.1 und 8.2 verdeutlichen dieses Array.

- der Operator `avl_knoten* operator () (myint i, myint jj)`, wobei i lokale und jj globale Indizes sind, liefert den Knoten, welcher den Eintrag (i, j) der frontalen Matrix repräsentiert. j ist der lokale Index, welcher $nodes(j) = jj$ genügt.

- der Operator `avl_knoten* operator () (myint k)`, wobei k lokaler Index ist, ist ein spezieller Fall des Operators `avl_knoten* operator () (myint i, myint jj)` falls $jj = nodes(i)$, also $i = j$. Dieser Operator wird verwendet, um auf Diagonaleinträge der frontalen Matrix zuzugreifen.

Parallel zum Array *nodes* gibt es einen Array *nodes_inv* der Länge n. Beide machen eine Konvertierung zwischen lokalen und globalen Indizes möglich. *nodes_inv* genügt $\forall j = 1, \ldots, m$ $nodes\_inv(nodes(j)) = j$. Im Unterschied zu *nodes*, ist das Array *nodes_inv* keine Membervariable. Es ist eine globale Variable, die in jedem Schritt (der multifrontalen Methode) aufdatiert werden muss, und zwar genau an den Positionen $(nodes(j))_{j=1:m}$ (Methode `invert_node_array()`). Bei unserer Beispielmatrix F sei z.B. $n = 10$. Dann sieht *nodes_inv* so aus ($*$ steht für irgendeinen Wert):

$$nodes\_inv = [*, *, *, *, *, 1, 2, *, 3, *]$$

Zuletzt die Routine `eliminate_node`. Sie ist die wichtigste und umfangreichste Routine des ILU-Teils des Programms:

```
void eliminate_node(ext_sqr_matr &F, myint piv_rc, avl_knoten
*pivot_node);
```

Der Index *piv_rc* ist lokal. Der Knoten *pivot_node*, welcher $pivot\_node = F(piv\_rc)$ genügt, ist der zu eliminierende Knoten. Die Routine generiert die Zeile und Spalte *piv_rc*, berechnet und datiert den condition estimator auf, berechnet die Spalte von L und die Zeile von U, wendet dropping an, berechnet den Rank-1 update nach der S oder T Variante, löscht die Spalte und Zeile *piv_rc* aus F und datiert anschließend die Members des Objekts F auf. F wird dann für weitere Assembly-Schritte gebraucht. In jedem Schritt $j = 1, 2, \ldots, n$ des multifrontalen Prozesses wird diese Routine höchstens zweimal aufgerufen, nämlich:

1. Keinmal, falls der gerade errechnete j-te Pivot nicht akzeptabel ist[2].

[2] das Akzeptanzkriterium wurde auf Seite 134 besprochen

2. Einmal, falls der gerade errechnete j-te Pivot akzeptabel ist und es nach seiner Elimination keinen akzeptablen verzögerten Pivot zu eliminieren gibt. Solche Pivots werden ausschließlich in der obere linken Ecke von F_j gesucht, denn nur dort könnte es Totally-Summed-Knoten geben, bei denen sich etwas geändert hat.

3. Zweimal, falls der gerade errechnete j-te Pivot akzeptabel ist, und es nach seiner Elimination einen akzeptablen verzögerten Pivot (in der obere linke Ecke von F_j) zu eliminieren gibt. Auf weitere Eliminierungen wird verzichtet.

Wird der Knoten j aus F_j eliminiert, so wird F_j nach dem Rank-1-Update zum U_j umbenannt.

Die Hauptvorteile des *ext_sqr_matr*-Formats gegenüber 2-dimensionalen Arrays und bekannten Sparse-Formaten sind:

1. seine Sparse-Fähigkeit
2. schneller Zugriff auf beliebige Einträge der Matrix
3. billiges Einfügen und/oder Löschen beliebiger Einträge
4. billiges Löschen ganzer Zeilen ohne Objekte löschen und/oder anlegen zu müssen
5. billiges Einfügen ganzer Zeilen, ohne Objekte zu löschen und/oder anlegen zu müssen

Der Nachteil dabei ist die Menge an Speicher, die dieses Format verbraucht. Jeder Knoten besitzt außer der unentbehrlichen `knoten_data` drei Zeigervariablen, was auf einer 32-bit-Maschine 12 Byte Extraspeicher bedeutet, und die `int` Variable `bal`.

Schließlich soll noch kurz gezeigt werden, wie man mittels meines Formats die Assembly-Phase billig implementiert:

$$F_j = \begin{pmatrix} A_{j,j} & A_{j,\mathcal{L}''_j} \\ A_{\mathcal{L}'_j,j} & 0 \end{pmatrix} \oplus U_{c_1} \oplus \ldots \oplus U_{c_s}$$

(hier ist $\mathcal{L}'_j = \text{Struct}(A_{j+1:n,j})$ und $\mathcal{L}''_j = \text{Struct}(A_{j,j+1:n})$).
Zuerst wird das *nodes* Array (für F_j) bestimmt:

$$nodes = \{j\} \cup \mathcal{L}'_j \cup \mathcal{L}''_j \cup U_{c_1}.nodes \cup \ldots \cup U_{c_s}.nodes$$

Diese Operation wird ausgeführt durch wiederholtes Anwenden der Funktion:

```
void unite(myint *a,myint len_a,myint *b, myint len_b, myint *c,
myint *ptr2len_c)
```

Sie liefert für zwei strikt aufsteigende Folgen von Indizes a der Länge len_a und b der Länge len_b, deren strikt aufsteigende Vereinigung c und seine Länge $*ptr2len\_c$ mit Aufwand $\mathcal{O}(len\_a + len\_b)$. Man braucht nur zwei Work-Arrays und keinerlei Verschiebungen.

Sei m die Länge von $nodes$. Die Teilbäume mit Wurzeln aus $c_1, \ldots, c_s$ sind knotendisjunkt. Daher gilt $dd = U_{c_1}.dd + \ldots + U_{c_s}.dd$. Ist j ein Blatt (beim Eliminationsbaum), so lege eine leere frontale Matrix[3] der Dimension m an und füge der Reihe nach den Eintrag $A_{j,j}$ und die Einträge aus $A_{\mathcal{L}'_j,j}$,$A_{j,\mathcal{L}''_j}$ ein. Ist j kein Blatt, so definiere F_j als das U_{c_t} mit der maximalen Anzahl von Nichtnullen $U_{c_t}.nnz$. Kleine Anpassungen an $U_{c_t}.nodes$ und $U_{c_t}.rows$ sind nötig. Falls j noch andere Söhne $(c_q)_q$ ausser c_t besitzt[4], dann füge deren Einträge der Reihe nach in F_j ein. Mache das Gleiche auch mit dem Eintrag $A_{j,j}$ und den Einträgen aus $A_{\mathcal{L}'_j,j}$,$A_{j,\mathcal{L}''_j}$. Diese Einfüge-Operationen brauchen folgende Funktion:

```
void sub_ind(myint *a, myint *b, myint len_b, myint *b_subind)
```

Sie liefert für zwei strikt aufsteigende Folgen von Indizes a und b, wobei $b \subset a$ und len_b die Länge von b ist, die Folge der Subindizes b_subind von a, so dass $b = a(b\_subind)$ mit Aufwand $\mathcal{O}(len\_a)$. Die Länge von a braucht man nicht explizit, die von b_subind muss $\geq len\_b$ genügen.

Durch diese Implementierung nimmt die Assembly-Phase unter 10% der ILU-Laufzeit in Anspruch. Die meiste Zeit, über 80%, nimmt jetzt erwartungsgemäß das Rank-1-Update in Anspruch.

Nach dem Assembly-Prozess werden die frontalen Matrizen $(U_{c_q})_{q \neq t}$ gelöscht und ihr Speicher freigegeben.

Es folgt ein Entwurf für die Multifrontal-ILU (siehe ewt. nächste Seite).

8.2 Implementierung ohne multifrontale Matrizen

Im Folgenden wird nur auf den Diagonalen pivotisiert.

Die verzögerte Eliminationstechnik bei der multifrontalen Methode ist keinesfalls ausschließlich auf die multifrontale Methode zugeschnitten. Man känn sie beim “ganz normalen“ Gauß mit Schur-Komplement anwenden. Im Grunde genommen ist es eine Pivotisierungstechnik wie die gewöhnliche Pivotisierung, die jeder kennt, nur etwas gewöhnungsbedürftiger. Außer dem Knoten, den wir nicht “mögen“, braucht eine verzögerte Elimination einen zweiten. In der Praxis ist dieser aber unbekannt und wird erst später feststehen. Um das Bild vollständig

[3] vom Typ `ext_sqr_matr`

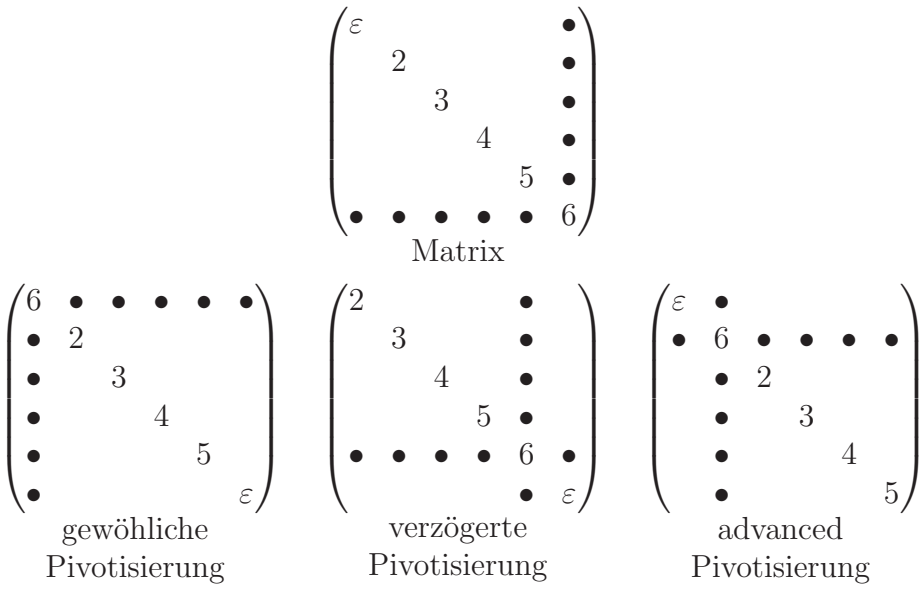
[4] dies wird selten der Fall sein, da jeder Knoten im Mittel $(n-1)/n$ Söhne hat

Algorithmus 8.1 ILU mit Multifrontal

Beschreibung: Input: A; Output π, ψ, L, D, U mit $P_\pi A Q_\psi^T \approx LDU$

$\pi = \psi = 1:n$, $k = 0$, $dd = 0$ {dd gibt die Gesamtanzahl der verzög. Knoten}
for $i = 1:n$ **do**
  bestimme die frontale Matrix F_i:
  seien $c_1, \ldots, c_S$ die Söhne von i in $T[A]$
  $F_i.nodes = \{j\} \cup \mathcal{L}'_j \cup \mathcal{L}''_j \cup U_{c_1}.nodes \cup \ldots \cup U_{c_s}.nodes$
  berechne $F_j = \begin{pmatrix} A_{j,j} & A_{j,\mathcal{L}''_j} \\ A_{\mathcal{L}'_j,j} & 0 \end{pmatrix} \oplus U_{c_1} \oplus \ldots \oplus U_{c_s}$
  $F_i.dd = U_{c_1}.dd + \ldots + U_{c_s}.dd$
  $F_i.dd{+}{+}$, $dd{+}{+}$ {es gilt $i == F_i.nodes(F_i.dd)$; i ist jetzt totally summed}
  if $dd > thresh$ **then** {Verzögerung "im Zaum" halten}
    return error: zu viele Knoten wurden verspätet
  end if
  if Knoten i akzeptabel als Pivot **then**
    $k{+}{+}$, $\pi(k) = \psi(k) = i$
    extrahiere $L_{:,k}$, D_k und $U_{k,:}$ aus F_i
    wende dropping an, datiere cond_est auf
    berechne rank-1 update
    lösche die Spalte und die Zeile dd aus F_i {$dd \to$ lokales Index}
    entferne i aus $F_i.nodes$ und "schließe die Lücke"
    $F_i.m{-}{-}$, $F_i.dd{-}{-}$, $dd{-}{-}$
    for $j = F_i.dd : -1 : 1$ **do** {versuche ≤ 1 verzögerte Knoten zu eliminieren}
      if Knoten $l = F_i.nodes(j)$ akzeptabel als Pivot **then**
        $k{+}{+}$, $\pi(k) = \psi(k) = l$
        Extrahiere $L_{:,k}$, D_k und $U_{k,:}$ aus F_i
        Wende dropping an, datiere cond_est auf
        berechne rank-1 update
        lösche die Spalte und die Zeile j aus F_i {$j \to$ lokaler Index}
        entferne l aus $F_i.nodes$ und "schließe die Lücke"
        $F_i.m{-}{-}$, $F_i.dd{-}{-}$, $dd{-}{-}$
        `break`
      end if
    end for
  end if
  $U_i = F_i$
end for
if $dd > 0$ **then** {bei U_n sind verzögerte Knoten übrig geblieben}
  finde eine LDU-Zerlegung, mit vollständiger Pivotisierung, für U_n
  übertrage diese Zerlegung in π, ψ, L, D, U
end if

zu machen, sei hier auch die Advanced Elimination (oder Pivotisierung) erwähnt. Folgende Abbildung zeigt alle drei Pivotisierungstechniken ([45]). Als unakzeptabler Pivot gilt hier der Knoten 1, und als zu permutierender Pivot dient der Knoten 6: (ε steht hier für einen sehr kleinen Wert, verglichen mit den anderen)



Advanced Elimination ist sinnvoll, wenn man 2×2 Pivots erlaubt, um die Symmetrie beizubehalten. Diese Pivotisierungstechnik stammt von Bunch und Kaufman ([18]). Die Abbildung zeigt, dass die Advanced Elimination den Fill-in erheblich erhöhen kann. Nicht so aber die verzögerte Pivotisierung. Sie zieht nur minimale Fill-ins nach sich. Der folgende Satz schätzt den Extra-Fill-in, welcher eine einzige verzögerte Pivotisierung verursacht (zufällige Nullen gibt es nicht) ab.

Satz 8.1 ([45]) *Sei A eine symmetrische $n \times n$ Matrix mit $G(A)$ zusammenhängend und $A = LDL^T$ ihre Cholesky-Zerlegung. Seien $1 \leq k < j \leq n$, so dass $k \in T[j]$. Verzögert man die Elimination von k unmittelbar hinter den Knoten j, so entsteht dadurch beim L ein Extra-Fill-in von*

$$\text{tiefe}\,(k) - \text{tiefe}\,(j) + \text{nnz}(L_{:,j}) - \text{nnz}(L_{:,k})$$

wobei $\text{tiefe}\,(*)$ *die Tiefe des Knoten $*$ in Eliminationsbaum $T[A]$ bezeichnet.*

Aus $\text{tiefe}\,(k) - \text{tiefe}\,(j) \leq j - k$, $\text{nnz}(L_{:,j}) \leq n - j + 1$ und $\text{nnz}(L_{:,k}) > 1$ folgt:

Korollar 8.1 *Unter den in Satz 8.1 beschriebenen Bedingungen ist das entstandene Extra-Fill-in immer kleiner als $(n - k)$.*

Die neue Implementierung geht ohne multifrontale Matrizen vor. Sie benutzt die "ganz normale" Gauß-Elimination mit Schur-Komplement und mit (ausschließ-

lich) verzögerter Pivotisierung. Arithmetisch gesehen, sind die Ergebnise identisch mit der multifrontalen Variante. Numerisch aber treten leichte Unterschiede auf. Folgender Algorithmus (evtl. nächste Seite) beschreibt die Arbeitsweise des Programms. Wegen der Array *nodes* gibt es auch hier lokale und globale Indizes. Ab jetzt nenne ich **mf_incomplete** die ILU mit Multifrontal und **not_mf** die zweite, neue Implementierung. Die Datentypen beim not_mf sind denen aus mf_incomplete sehr ähnlich, die beschrieben wurden. Die wichtigsten Unterschiede sind:

- Beim mf_incomplete werden viele frontale Matrizen gebraucht, beim not_mf nur eine.

- Die j-te Spalte einer frontalen Matrix wurde beim mf_incomplete generiert, indem man in jeder Zeile nach einem Element j sucht. Bei not_mf, sind die Zeilen auch als AVL-Bäume gespeichert, die Spalten als doppelt verkettete Listen von avl_knotent.

- Beim mf_incomplete ist es unmöglich, eine switch-to-full anzuwenden, weil das Schur-Komplement nicht in geschlossener Form vorliegt (es ist zerstreut über die frontalen Matrizen). Dadurch wird in den letzten (einige Hundert) Schritten Speicher und Zeit verschwendet. Beim not_mf ist ein switch-to-full eingebaut.

Der Datentyp `datensatz` enthält hier zwei Members mehr, `avl_knoten *next_col` und `avl_knoten *prev_col` um einen AVL-Knoten in die doppelt verkettete Liste seiner Spalten zu integrieren:

```
struct datensatz
{
   key_typ col_ind; //global Index, dient gleichzeitig auch als
Knotenschlüssel
   key_typ row_ind; //global Index
   avl_knoten *next_col;
   avl_knoten *prev_col;
   double val;
};
typedef avl_knoten *avl_baum;
typedef avl_knoten *avl_knoten_liste; // Listen für die Spalten
```

Bei der Definition von `struct avl_knoten` ändert sich nichts. Folgende drei Methoden musste ich an die `namespace avl_operations` hinzufügen, um mit den Spaltenlisten umzugehen:

Algorithmus 8.2 ILU mit verzögerte Pivotisierung

Beschreibung: Input: A; Output π, ψ, L, D, U mit $P_\pi A Q_\psi^T \approx LDU$

$F = A$ {der Datentyp von F wird unten besprochen}
$nodes = 1 : n$, $dd = 0$ {dd ist die Anzahl der bisher verzögerten Knoten}
$m = n$ {m wird die (relevante) Länge des $nodes$ sein}
$\pi = \psi = 1 : n$, $k = 0$
for $i = 1 : n$ **do**
  $dd++$, {gilt $i == nodes(dd)$; betrachte i als totally summed}
  if $dd > thresh$ **then** {Verzögerung "im Zaum" halten}
    return error: zu viele Knoten wurden verspätet
  end if
  if Knoten i akzeptabel als Pivot **then**
    $k++$, $\pi(k) = \psi(k) = i$
    Extrahiere $L_{:,k}$, D_k und $U_{k,:}$ aus F
    Wende dropping an, datiere cond_est auf
    berechne rank-1 update
    lösche die Spalte und die Zeile dd aus F {$dd \rightarrow$ lokales Index}
    entferne i aus $nodes$ und "schließe die Lücke"
    $m--$, $dd--$
    for $j = dd : -1 : 1$ **do** {versuche ≤ 1 verzögerte Knoten zu eliminieren}
      if Knoten $l = nodes(j)$ akzeptabel als Pivot **then**
        $k++$, $\pi(k) = \psi(k) = l$
        Extrahiere $L_{:,k}$, D_k und $U_{k,:}$ aus F
        Wende dropping an, datiere cond_est auf
        berechne rank-1 update
        lösche die Spalte und die Zeile j aus F {$j \rightarrow$ lokales Index}
        entferne l aus $nodes$ und "schließe die Lücke"
        $m--$, $dd--$
        `break`
      end if
    end for
  end if
end for
if $dd > 0$ **then** {es sind verzögerte Knoten übrig geblieben}
  finde eine LDU-Zerlegung, mit vollständigen Pivotisierung, für F
  übertrage diese Zerlegung in π, ψ, L, D, U
end if

```
void avl_knoten_liste2array(avl_knoten_liste lis, avl_knoten
**knoten_array, myint *ptr2size);
inline void in_COL_liste_einfuegen(avl_knoten_liste *pliste,
avl_knoten *neu_knoten);
inline void aus_COL_liste_entfernen(avl_knoten_liste *pliste,
avl_knoten *nyje);
```

Der Datentyp für die Matrix:

```
nodes_inv=new myint[n]; // n ist die Dimension des Problems
struct schur_avl_matr
{
   myint n;
   myint nr_elim;
   myint m;
   myint dd;
   myint elimORdelayed; // elimORdelayed = nr_elim+dd
   myint nnz;
   myint *orig_nodes;
   myint *nodes;
   avl_baum *orig_rows;
   avl_baum *rows;
   avl_knoten_liste *orig_cols;
   avl_knoten_liste *cols;

   schur_avl_matr(myint n_inp)
   {
      nr_elim=0; m=n; dd=0; elimORdelayed=nr_elim+dd; nnz=0;
      orig_nodes=prgres_aritm( n, 1); // erzeugt die Array 1:n
      nodes=orig_nodes+nr_elim;
      zeros(&orig_rows,n); // leere Zeilen
      rows=orig_rows+nr_elim;
      zeros(&orig_cols,n); // leere Spalten
      cols=orig_cols+nr_elim;
   }
   avl_knoten* operator () (myint i, myint j)
   { // i,j sind lokale Indizes
      return avl_operations::avl_suchen(aks(nodes,j),aks(rows,i));
   }
(es geht weiter)
```

```
  avl_knoten* operator () (myint k);
  { // k ist lokale Index
    return avl_operations::avl_suchen(aks(nodes,k),aks(rows,k));
  }
  void neue_element_einfuegen(myint i, myint j, double val)
  { // i,j sind lokale Indizes
    datensatz d; d.val=val;
    d.row_ind=aks(nodes,i); // global
    d.col_ind=aks(nodes,j); // global
    // füge in die Zeile ein
    avl_knoten *neu_knoten=avl_operations::avl_einfuegen(d,rows+i-1);
    // füge am Anfang der Spaltenliste
    avl_operations::in_COL_liste_einfuegen(cols+j-1,neu_knoten);
    nnz++; // inkremeniere nnz
  }
  void invert_node_array()
  { // invertiert nur den Bereich 1:dd von nodes
    for(myint j=1; j<=dd; j++)
      aks(nodes_inv,aks(nodes,j))=j;
  }
  myint inverse_node( myint jj )
  { // jj ist globaler Index, liefert den entspr. lokalen Index j
    return ((jj>elimORdelayed)?(jj-nr_elim):aks(nodes_inv,jj));
  }
  double fuellung()
  {
    return (((double)nnz)/m)/m;
  }
  ~schur_avl_matr()
  {
    for(myint k=0; k<m; k++)
      avl_operations::avl_deallocate(rows+k);
    free(orig_nodes);
    free(orig_rows);
    free(orig_cols);
  }
};
```

Wie beim mf_incomplete, gibt es die Hauptfunktion:

```
void eliminate_node_sparse(schur_avl_matr &F,myint piv_rc,avl_knoten
*pivot_node)
```

Sie verrichtet genau das, was die Funktion `void eliminate_node (ext_sqr_matr &F, myint piv_rc, avl_knoten *pivot_node)` beim mf_incomplete verrichtete

(siehe Beschreibung dort). Man muss hier aufpassen, die durch den Rank-1-Update neu entstandenen Knoten in deren Spaltenlisten hinzuzufügen.

Folgende Abbildungen verdeutlichen die Rollen der vorgestellten Variablen:

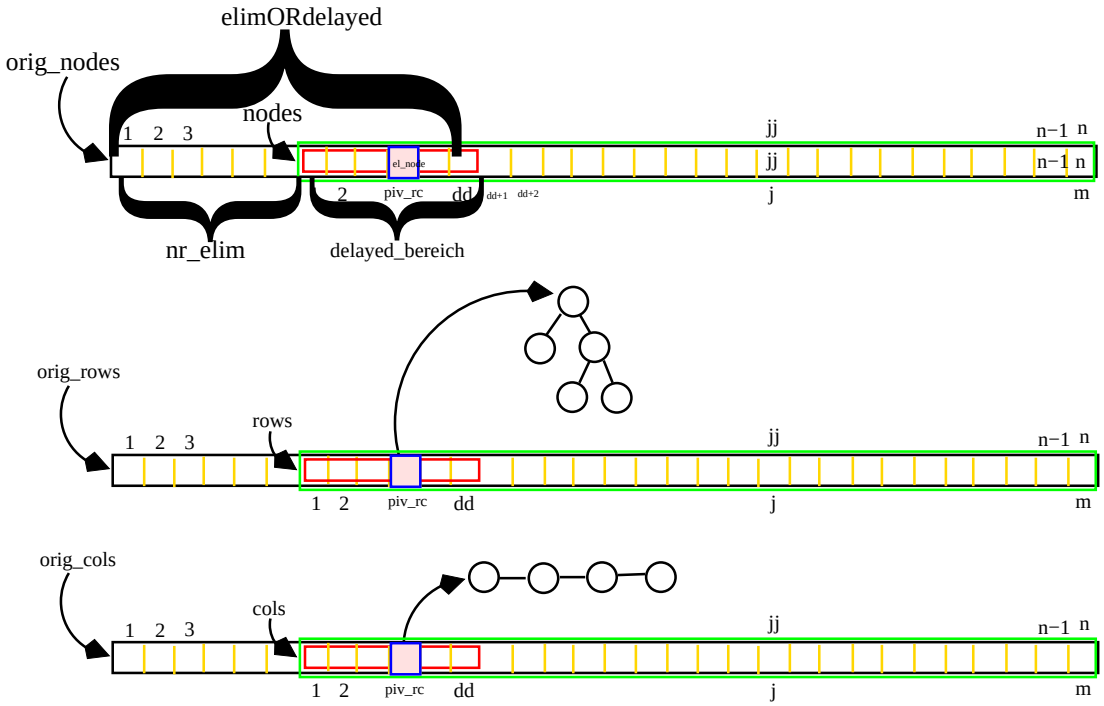


Abbildung 8.3: Veranschaulichung der Variablen

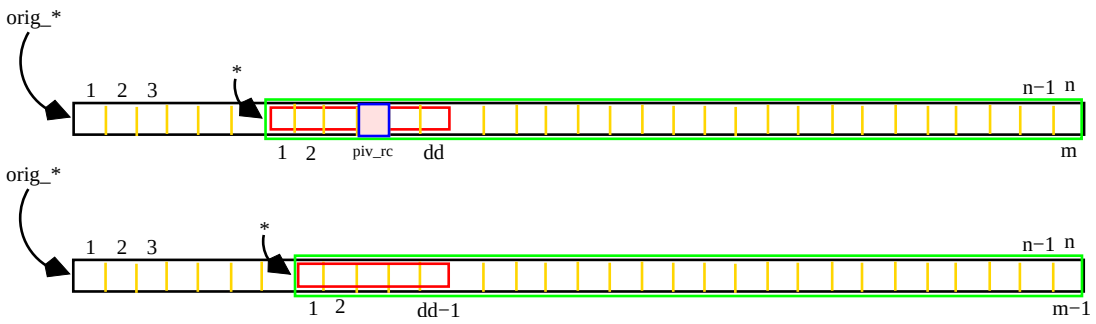


Abbildung 8.4: nach Elimination, “Lücke“ schließen durch Verschiebung

8.3 Beide Implementierungen: Vor- und Nachteile

Wir stellen hier die Vor- und Nachteile von mf_incomplete und not_mf in Stichpunkten vor.

- Bei mf_incomplete werden die Spalten nicht explizit gespeichert; bei not_mf werden sie als doppelt verkettete Listen gespeichert.
- Die nicht explizite Speicherung der Spalten bei mf_incomplete, ermöglicht uns die Parallelisierung seiner Rank-1-Updates.
- Bei mf_incomplete ist es schwer, eine "switch to full" einzubauen; bei not_mf ist es einfach (und von mir auch realisiert). Um die aktuelle Füllung (bei not_mf) zu bestimmen, reicht ein Aufruf der $\mathcal{O}(1)$-Memberfunktion `double fuellung()`.
- Bei beiden Versionen ist es einfach, Einträge in die Matrix einzufügen und/oder zu löschen.
- Bei beiden Versionen ist es einfach, eine bestimmte Zeile und/oder Spalte aus der Matrix zu extrahieren und zu löschen.
- Mit einen Trick wurde die Assembly-Phase beim mf_incomplete sehr billig; bei not_mf besteht dieses Problem nicht.
- Bei mf_incomplete liegt die Schur Matrix über die frontalen Matrizen zerstreut; bei not_mf ist diese Matrix explizit gespeichert. Die frontalen Matrizen könnten überlappende Einträge haben. Durch diese "Mehrfachspeicherungen" ist zu erwarten, dass die mf_incomplete-Version mehr Speicher als not_mf braucht.
- Bei mf_incomplete muss man eine irreduzible Matrix als Eingabe übergeben; da not_mf nicht vom Eliminationsbaum abhängt (Man ist nur auf die Blätterknoten angewiesen. Die sind auch ohne den Eliminationsbaum einfach zu identifizieren nach der Bemerkung 4.1), ist not_mf leicht auf beliebige Matrizen mit nullfreier Diagonale erweiterbar.
- Bei beiden Implementierungen ist die lokal-globale Konvertierung der Indizes in $\mathcal{O}(dd)$[5] realisierbar.
- Bei mf_incomplete ist die "Lückenschließung" etwas aufwendiger als bei not_mf, wo sie in $\mathcal{O}(dd)$ realisierbar ist.

[5] $\mathcal{O}(dd)$ kann in der Regel als $\mathcal{O}(1)$ angesehen werden

- Bei beiden Implementierungen ist der große Speicherverbrauch ein Nachteil, verglichen mit einer reinen Listenimplementierung. Das ist der Preis, der für die flexiblen Matrixformate zu zahlen ist.

- Bei mf_incomplete geht eine gute Eigenschaft der multifrontalen Methode zur Berechnung der vollständigen LU-Faktorisierung verloren: Die Möglichkeit, Level 2 BLAS beim Schur-Update zu verwenden (dort entstehen kleine, aber volle, quadratische Matrizen)

Aus diesen Überlegungen ist zu erwarten, dass die not_mf-Variante schneller als die mf_incomplete-Variante ist. Siehe auch die Vergleiche aus der Tabellen 8.1 und 8.2.

8.4 Ergebnisse und Vergleiche

Im Abschnitt 6.1 sahen wir, dass es zwei Möglichkeiten gibt, das Rank-1-Update bei Gauß-Elimination für ILU's zu bestimmen. Dementsprechend entstehen, für jede der Varianten mf_incomplete und not_mf, zwei Untervarianten. Ich habe sie **mf_S** und **mf_T** für die mf_incomplete-Variante und **nmf_S** und **nmf_T** für die not_mf-Variante benannt. Alle vier Varianten wurden gründlich getestet. Dabei hat sich die nmf_S-Variante als die beste erwiesen.

Alle vier Varianten gehen gleich vor, abgesehen von der die ILU bestimmenden (Haupt-)Routine. Da ruft jede ihre eigene. Detailliert wurde die Vorgehensweise in Abschnitt 7.3 beschrieben. Einen kleinen Unterschied gibt es für die beiden Untervarianten nmf_S und nmf_T. Diese brauchen als Eingabe einen Extra-Parameter, um den Zeitpunkt für das switch-to-full festzulegen. Als Standardwert benutze ich 0.15, d.h. sobald die Füllung nnz die Schwelle $0.15 \cdot m^2$ überschreitet, wird auf full-mode umgeschaltet. Ich habe verschiedene Schwellenwerte getestet. Die besten Ergebnise wurden bei 0.15 erzielt.
Ein Aufruf z.B. der Routine nmf_S für die Matrix af23560 sieht so aus:

```
make run2S fill_rate=5 piv_tol=1e-1 tau=0.4
nmf_S.out 5 1e-1 0.4 300 1e-08 0.15 30 17 1.5e-08
 ILU: Gauss mit delayed nodes und avl
 GMRES: split_gmres, implementiert mit blas und sparse_blas
 starke_komp_anzahl = 1
 Dominante Komponente besitzt Dimension: 23560
 Matrix : (23560 X 23560)
 'MPD_Matching' und 'imatrix' Zeitdauer: 2.103460e-01 sec.
 'amd' Anordnung Zeitdauer: 1.600020e-01 sec.
 Nach MinDeg: fill_in upper bound = 18.089041
```

```
ILU Parameter:  fill_rate=5.0, piv_tol=1.00e-01,
                tau=4.00e-01, sparse2full_factor=0.150
SERIELL, Schur-update: S-Variant

Es wurden insgesamt 5 Knoten 'verspaetet'.
Von denen wurden 5 'rechtzeitig' eliminiert.
fill_in-faktor: 1.174649

'ILU' Zeitdauer: 1.598570e+00 sec.

Starte GMRES(30). Berechnungen in Norm 2.
Anfangsresiduum des Ursprungssystems: 8.835135e+01
Zielresiduum des Ursprungssystems: 1.325270e-06

Schritt: 0, norm(r_y)=8.462035e+01,  norm(r_x)=8.835135e+01
Schritt: 1, norm(r_y)=1.007906e+02,  norm(r_x)=1.632460e+02
Schritt: 2, norm(r_y)=8.449155e+01,  norm(r_x)=1.066017e+02
Schritt: 3, norm(r_y)=3.184607e+01,  norm(r_x)=5.744217e+01
Schritt: 4, norm(r_y)=7.643825e+00,  norm(r_x)=1.242618e+01
Schritt: 5, norm(r_y)=1.830534e+00,  norm(r_x)=3.987974e+00
Schritt: 6, norm(r_y)=2.940341e-01,  norm(r_x)=3.940075e-01
Schritt: 7, norm(r_y)=6.633732e-02,  norm(r_x)=1.232219e-01
Schritt: 8, norm(r_y)=1.425178e-02,  norm(r_x)=1.940439e-02
Schritt: 9, norm(r_y)=2.872616e-03,  norm(r_x)=5.660283e-03
Schritt: 10, norm(r_y)=6.981254e-04,  norm(r_x)=1.367960e-03
Schritt: 11, norm(r_y)=1.365036e-04,  norm(r_x)=1.820300e-04
Schritt: 12, norm(r_y)=3.239420e-05,  norm(r_x)=6.141936e-05
Schritt: 13, norm(r_y)=7.877747e-06,  norm(r_x)=1.487394e-05
Schritt: 14, norm(r_y)=1.344596e-06,  norm(r_x)=2.411535e-06
Schritt: 15, norm(r_y)=3.512715e-07,  norm(r_x)=7.555189e-07

GMRES(30) konvergierte in 15 Schritten
Zielresiduum: 1.325270e-06 wurde erreicht: norm(r_x)=7.555189e-07
GMRES Zeitdauer: 4.005525e+00 sec.
```

Die nächsten zwei Tabellen vergleichen meine eigenen vier Routinen **mf_S**, **mf_T**, **nmf_S** und **nmf_T** untereinander. Alle Tests wurden auf eine P4 HT 2.8GHz Maschine unter Linux Suse 10.0 ausgeführt. Kompiliert wurde mit den -O3 Option. Das Zeichen – an der Tabelle steht für Versagen. Zeitangaben sind in Sekunden. Alle Testmatrizen stammen aus [23] und [2]. Für eine Diskussion der Ergebnise siehe Abschnitt 8.3.

Matrix Nr.	Matrix Name	Dim (dominant)	Preproc. Dauer	fill-in approx.	Aufgerufene Funktion	Hauptfunkt. Dauer (inklusive GMRES)	ILU fill-in	Aufgerufene Funktion	Hauptfunkt. Dauer (inklusive GMRES)	ILU fill-in
1	af23560	23560	0.37	18.09	mf_S	6.84	1.18	mf_T	12.58	1.13
2	aft01	8205	0.05	4.54	mf_S	1.20	1.59	mf_T	1.09	1.25
3	alemdar	6240	0.06	17.40	mf_S	3.00	12.18	mf_T	3.32	11.70
4	av41092	41086	7.43	7.64	mf_S	48.70	2.20	mf_T	137.70	2.39
5	bayer10	10803	0.10	8.26	mf_S	0.40	0.77	mf_T	0.38	0.82
6	bcsstk18_1	11066	0.15	8.89	mf_S	1.69	1.02	mf_T	2.53	0.76
7	c-62	41729	1.51	90.45	mf_S	5.09	0.44	mf_T	11.93	0.34
8	c-70	68924	1.08	48.11	mf_S	8.48	1.08	mf_T	292.53	2.28
9	cavity24	4241	0.09	2.74	mf_S	0.71	0.92	mf_T	1.16	0.73
10	cfd1	70656	1.09	39.26	mf_S	-	-	mf_T	1473.3	5.12
11	circuit_3	7607	0.05	1.46	mf_S	0.17	1.12	mf_T	0.23	0.99
12	circuit_4	52005	0.27	1.66	mf_S	3.23	0.89	mf_T	1.26	0.87
13	dw8192	8192	0.06	11.65	mf_S	1.01	3.76	mf_T	1.51	2.70
14	epb3	84617	0.33	10.84	mf_S	9.98	1.37	mf_T	13.22	1.14
15	ex19	5850	0.08	2.66	mf_S	2.28	2.61	mf_T	1.62	1.92
16	fidapm37	5742	0.23	2.76	mf_S	17.60	2.51	mf_T	17.47	1.67
17	garon2	13535	0.18	7.05	mf_S	3.78	1.14	mf_T	8.87	1.05
18	gemat12	4552	0.02	1.6	mf_S	0.40	1.94	mf_T	0.32	2.06
19	goodwin	7319	0.27	3.96	mf_S	4.12	1.70	mf_T	6.40	1.27
20	graham1	8398	1.48	4.31	mf_S	5.59	1.93	mf_T	6.97	1.40
21	hcircuit	92144	0.30	1.24	mf_S	3.24	0.69	mf_T	1.21	0.74
22	igbt3	10938	0.11	8.05	mf_S	1.57	1.94	mf_T	-	-
23	jan99jac120sc	36070	1.08	21.64	mf_S	5.29	1.99	mf_T	10.54	2.04
24	jan99jac120	36070	1.09	21.64	mf_S	6.19	2.11	mf_T	11.36	2.44
25	mark3jac140	61849	2.58	61.13	mf_S	159.34	17.76	mf_T	271.80	13.16
26	memplus	17736	0.07	1.24	mf_S	0.80	0.70	mf_T	0.30	0.66
27	nasasrb	54870	1.00	8.91	mf_S	-	-	mf_T	435.63	3.88
28	onetone1	32211	0.28	11.65	mf_S	18.91	3.52	mf_T	23.92	1.70
29	onetone2	32211	0.19	9.58	mf_S	7.32	3.38	mf_T	12.13	2.10
30	osreg_1	2205	0.01	10.91	mf_S	0.12	1.00	mf_T	0.12	1.00
31	Pres_Poisson	14822	0.26	7.00	mf_S	8.78	1.37	mf_T	23.04	1.13
32	rma10	46828	0.96	4.05	mf_S	30.60	1.32	mf_T	48.70	0.97
33	twotone	105740	0.71	11.07	mf_S	162.02	6.24	mf_T	205.54	5.25
34	venkat01	62424	0.73	6.72	mf_S	6.07	0.16	mf_T	6.23	0.16
35	venkat50	62424	0.67	6.71	mf_S	29.39	2.33	mf_T	61.41	1.63
36	wang4	26068	0.17	59.46	mf_S	1.72	1.08	mf_T	2.29	0.94

Tabelle 8.1: *mf_S*, *mf_T*, *nmf_S* und *nmf_T* untereinander im Vergleich (es geht weiter)

Matrix Nr.	Matrix Name	Aufgerufene Funktion	Hauptfunkt. Dauer (inklusive GMRES)	ILU fill-in	Aufgerufene Funktion	Hauptfunkt. Dauer (inklusive GMRES)	ILU fill-in
1	af23560	nmf_S	5.60	1.17	nmf_T	12.08	1.14
2	aft01	nmf_S	1.14	1.59	nmf_T	1.06	1.25
3	alemdar	nmf_S	2.90	12.16	nmf_T	3.21	11.66
4	av41092	nmf_S	37.31	2.18	nmf_T	96.60	2.40
5	bayer10	nmf_S	0.38	0.77	nmf_T	0.36	0.82
6	bcsstk18_1	nmf_S	1.57	1.02	nmf_T	2.05	0.76
7	c-62	nmf_S	3.48	0.44	nmf_T	6.26	0.34
8	c-70	nmf_S	5.82	1.08	nmf_T	134.29	2.26
9	cavity24	nmf_S	0.67	0.94	nmf_T	1.02	0.74
10	cfd1	nmf_S	-	-	nmf_T	1661.9	5.12
11	circuit_3	nmf_S	0.26	1.15	nmf_T	0.18	0.99
12	circuit_4	nmf_S	3.09	0.89	nmf_T	1.16	0.87
13	dw8192	nmf_S	1.00	3.79	nmf_T	1.15	2.66
14	epb3	nmf_S	7.99	1.55	nmf_T	12.82	1.14
15	ex19	nmf_S	1.93	2.61	nmf_T	1.45	1.92
16	fidapm37	nmf_S	16.21	2.51	nmf_T	15.37	1.67
17	garon2	nmf_S	3.40	1.15	nmf_T	9.86	0.96
18	gemat12	nmf_S	0.47	1.94	nmf_T	0.39	2.06
19	goodwin	nmf_S	4.89	1.69	nmf_T	6.69	1.26
20	graham1	nmf_S	4.54	1.92	nmf_T	5.29	1.43
21	hcircuit	nmf_S	3.18	0.69	nmf_T	1.39	0.74
22	igbt3	nmf_S	1.63	1.95	nmf_T	-	-
23	jan99jac120sc	nmf_S	5.39	1.94	nmf_T	8.52	2.05
24	jan99jac120	nmf_S	5.76	2.10	nmf_T	10.80	2.36
25	mark3jac140	nmf_S	143.04	17.04	nmf_T	267.59	14.13
26	memplus	nmf_S	0.79	0.71	nmf_T	0.30	0.66
27	nasasrb	nmf_S	-	-	nmf_T	493.23	3.88
28	onetone1	nmf_S	12.88	3.51	nmf_T	16.64	1.70
29	onetone2	nmf_S	7.32	3.38	nmf_T	11.15	2.09
30	osreg_1	nmf_S	0.10	1.00	nmf_T	0.10	1.00
31	Pres_Poisson	nmf_S	8.20	1.33	nmf_T	24.28	1.13
32	rma10	nmf_S	31.18	1.32	nmf_T	55.69	0.87
33	twotone	nmf_S	39.18	6.22	nmf_T	38.38	5.01
34	venkat01	nmf_S	5.23	0.16	nmf_T	5.29	0.16
35	venkat50	nmf_S	27.66	2.31	nmf_T	63.88	1.40
36	wang4	nmf_S	1.18	1.08	nmf_T	1.72	0.94

Tabelle 8.2: *mf_S*, *mf_T*, *nmf_S* und *nmf_T* untereinander im Vergleich

Als nächstes vergleiche ich meine Routinen mit ILUPACK_V1.1 ([17],[13]) von Bollhöfer (und Saad) mit standard Optionen:

- preproc. initial system: scaling + ddPQ ordering
- reordering subsystems: scaling + ddPQ ordering

und das von David Fritzsche über ein Makefile erzeugte ILUPACK_MMD mit Standard-Optionen:

- preproc. initial system: scaling + MMD ordering
- reordering subsystems: scaling + MMD ordering

(MMD = Multiple Minimum Degree ist eine Implementierung des MinDeg Algorithmus von Liu).

Folgende Tabelle enthält die Ergebnise der durchgeführten Tests. Alle Tests wurden auf einer P4 HT 2.8GHz Maschine unter Linux Suse 10.0 ausgeführt. Kompiliert wurde mit der -O3 Option. Das Zeichen − an der Tabelle steht für Versagen. Zeitangaben sind in Sekunden. Die Testmatrizen sind die gleichen wie oben. Abbildungen 8.5 und 8.6 stellen den Inhalt diese Tabelle als Plots detaillierter dar.

Es sei hier erwähnt, dass die (ausführbare binary) ILUPACK_V1.1, die ich getestet habe, nicht selbst erzeugt ist (über ein Makefile), sondern diejenige, die beim ILUPACK_V1.1 defaultmäßig dabei liegt und (höhst wahrscheinlich) von Bollhöfer selbst kompiliert wurde. Es folgt eine kurze Beschreibung dieser Methode.

Matrix Nr.	Matrix Name	Dim (dominant)	Preproc. Dauer	fill-in approx.	Aufgerufene Funktion	Hauptfunkt. Dauer (inklusive GMRES)	ILU fill-in	ilupack_MMD Dauer (inklusive GMRES)	ilupack_MMD fill-in	ilupack_V1.1 Dauer (inklusive GMRES)	ilupack_V1.1 fill-in
1	af23560	23560	0.37	18.09	nmf_S	5.60	1.17	7.3	1.1	8.1	1.4
2	aft01	8205	0.05	4.54	nmf_S	1.14	1.59	1.5	1.3	1.8	1.0
3	alemdar	6240	0.06	17.40	nmf_S	2.90	12.16	1.48	9.7	6.9	17.5
4	av41092	41086	7.43	7.64	nmf_S	37.31	2.18	155.0	5.7	17.7	0.8
5	bayer10	10803	0.15	8.89	nmf_S	0.38	0.77	1.06	5.2	0.28	2.0
6	bcsstk18_1	11066	0.15	8.89	nmf_S	1.57	1.02	2.7	0.6	2.6	1.1
7	c-62	41729	1.51	90.45	nmf_S	3.48	0.44	-	-	34.0	6.6
8	c-70	68924	1.08	48.11	nmf_S	5.82	1.08	-	-	13.3	1.0
9	cavity24	4241	0.09	2.74	nmf_S	0.67	0.94	-	-	1.52	0.9
10	cfd1	70656	1.40	39.26	nmf_T	1661.9	5.12	238	6.2	360	11.0
11	circuit_3	7607	0.05	1.46	nmf_T	0.18	0.99	0.7	1.2	0.18	1.0
12	circuit_4	52005	0.27	1.66	nmf_T	1.16	0.87	14.61	0.9	1.73	1.1
13	dw8192	8192	0.06	11.65	nmf_T	1.15	2.66	1.78	1.9	2.38	1.6
14	epb3	84617	0.33	10.84	nmf_S	7.99	1.55	18.2	0.9	17.74	0.8
15	ex19	5850	0.08	2.66	nmf_T	1.45	1.92	-	-	-	-
16	fidapm37	5742	0.23	2.76	nmf_T	15.37	1.67	-	-	-	-
17	garon2	13535	0.18	7.05	nmf_S	3.40	1.15	5.71	0.9	5.1	0.9
18	gemat12	4552	0.02	1.6	nmf_S	0.47	1.94	5.52	15.0	0.72	1.3
19	goodwin	7319	0.27	3.96	nmf_S	4.89	1.69	-	-	4.00	1.6
20	graham1	8398	1.48	4.31	nmf_S	4.54	1.92	-	-	3.7	1.5
21	hcircuit	92144	0.30	1.24	nmf_S	3.18	0.69	15.5	0.6	9.05	0.7
22	igbt3	10938	0.11	8.05	nmf_S	1.63	1.95	2.13	1.0	1.47	1.1
23	jan99jac120sc	36070	1.08	21.64	nmf_S	5.39	1.94	10.1	1.2	1.80	1.5
24	jan99jac120	36070	1.09	21.64	nmf_S	5.76	2.10	34.75	6.7	5.9	5.9
25	mark3jac140	61849	2.58	61.13	nmf_S	143.04	17.04	-	-	-	-
26	memplus	17736	0.07	1.24	nmf_T	0.30	0.66	0.81	0.4	1.13	0.7
27	nasasrb	54870	1.00	8.91	nmf_T	493.23	3.88	56.9	6.3	390	13.0
28	onetone1	32211	0.28	11.65	nmf_T	16.64	1.7	-	-	3.3	1.8
29	onetone2	32211	0.19	9.58	nmf_T	11.15	2.09	-	-	2.06	1.7
30	osreg_1	2205	0.01	10.91	nmf_S	0.10	1.00	0.12	0.4	0.11	0.7
31	Pres_Poisson	14822	0.26	7.00	nmf_S	8.20	1.33	10.2	1.3	10.6	1.7
32	rma10	46828	0.96	4.05	nmf_S	31.18	1.32	-	-	41.0	1.5
33	twotone	105740	0.71	11.07	nmf_T	38.38	5.01	-	-	24.10	2.6
34	venkat01	62424	0.73	6.72	nmf_S	5.23	0.16	9.7	0.1	8.8	0.3
35	venkat50	62424	0.67	6.71	nmf_S	27.66	2.31	24	1.8	40.0	2.2
36	wang4	26068	0.17	59.46	nmf_S	1.18	1.08	3.17	0.4	2.11	0.4

Tabelle 8.3: *nmf_S* und *nmf_T* im Vergleich mit *ilupack_MMD* und *ilupack_V1.1*

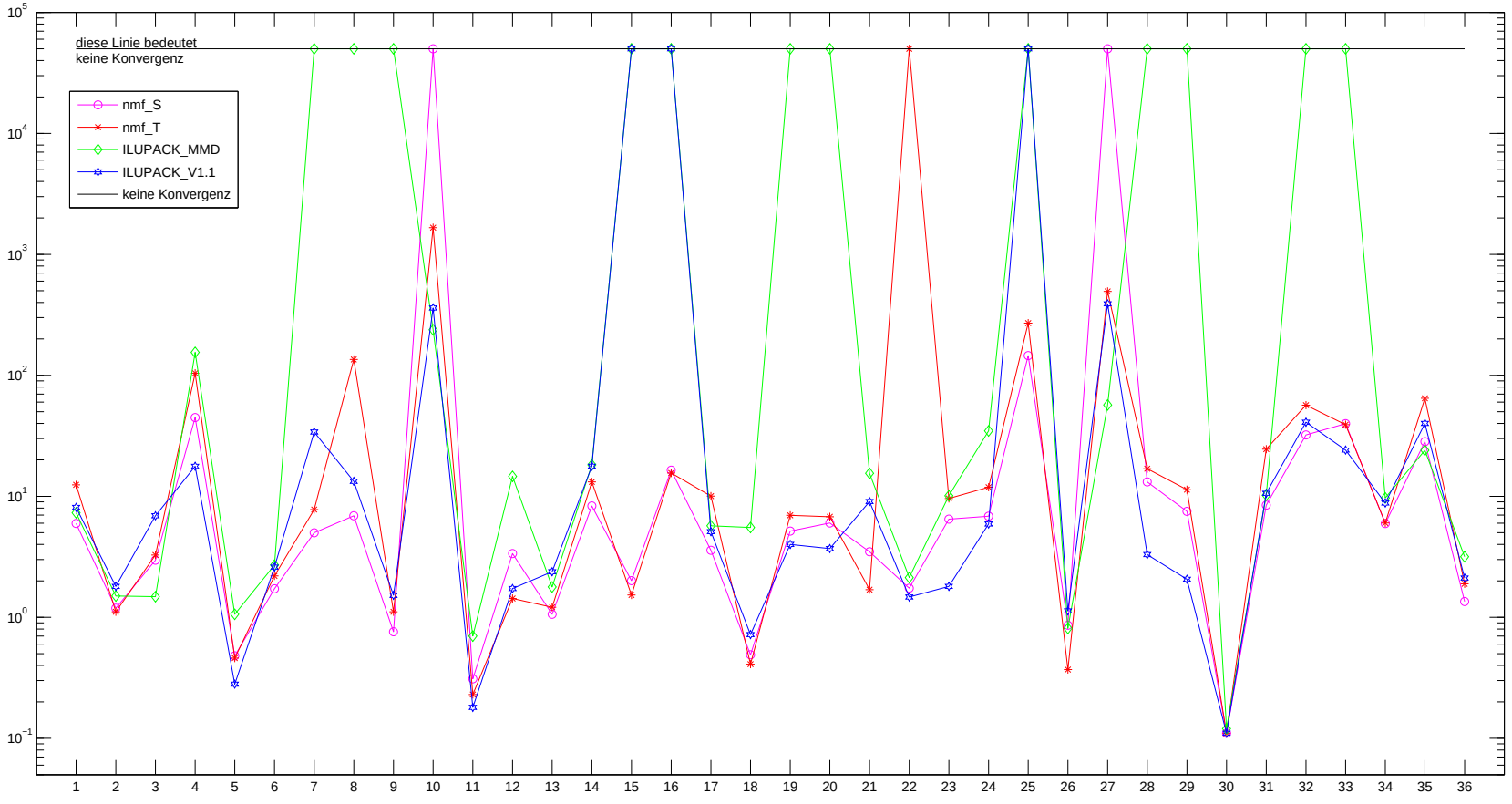


Abbildung 8.5: Die Ausführungszeiten von *nmf_S*, *nmf_T*, *ilupack_MMD* und *ilupack_V1.1*

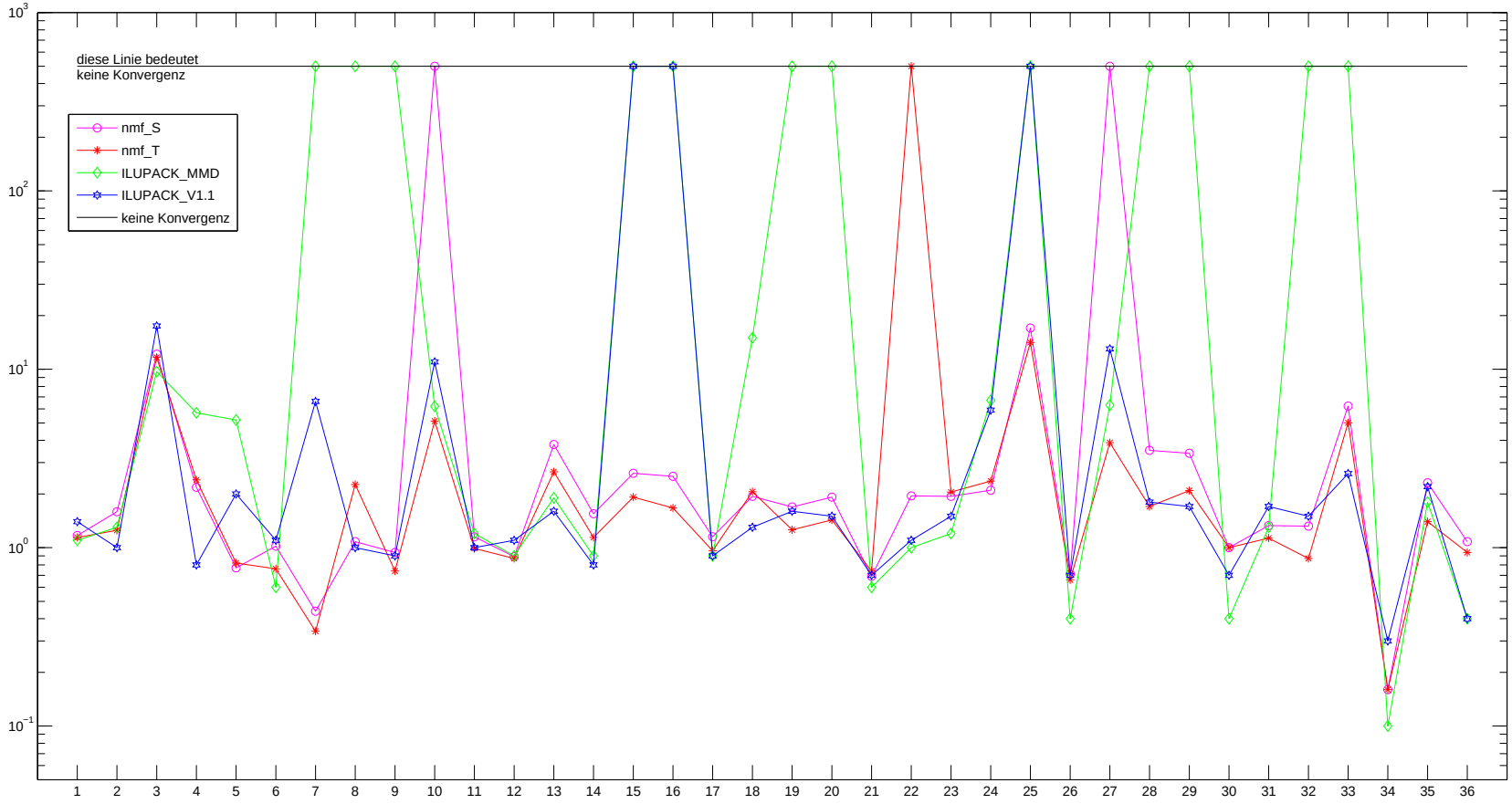


Abbildung 8.6: Die Fill-in's von *nmf_S*, *nmf_T*, *ilupack_MMD* und *ilupack_V1.1*

ILUPACK bestimmt ein Präkonditionierer des Typs multilevel, ähnlich wie ARMS von Saad und Suchomel ([62]). Solche Präkonditionierer arbeiten rekursiv. Zuerst läuft ILUPACK über drei Schritte:

1. Der ddPQ Anordnung ([59]) liefert für eine Matrix A zwei Permutationen P,Q, so dass:
$$PAQ^T = \begin{pmatrix} B & F \\ E & C \end{pmatrix} \tag{8.1}$$
wobei B eine gewisse diagonale Dominanz aufweist und seine Elimination wenig fill-in nach sich zieht.

2. Mittels der Crout-Doolittle-Variante wird versucht, die Zeilen und Spalten des B-Blocks, nur die und nicht mehr, zu eliminieren. Dabei geht man wie bei der Abbildung 8.7 vor. Es wird ausschließlich in die Hauptdiagonale pivotisiert. Wird ein Pivot als nicht akzeptabel eingestuft, so wird er am Ende der Hauptdiagonale permutiert[6]. Alle Rank-1-Updates werden nach den **S-Variante** durchgeführt. Dadurch erhält man:
$$\widehat{P}A\widehat{Q}^T = \widetilde{P}(PAQ^T)\widetilde{P}^T = \widetilde{P}\begin{pmatrix} B & F \\ E & C \end{pmatrix}\widetilde{P}^T =$$
$$= \left(\begin{array}{c|cc} B_{1,1} & B_{1,2} & F_1 \\ \hline B_{2,1} & B_{2,2} & F_2 \\ E_1 & E_2 & C \end{array}\right) = \left(\begin{array}{c|c} \widehat{B} & \widehat{F} \\ \hline \widehat{E} & \widehat{C} \end{array}\right) \approx \begin{pmatrix} L_{\widehat{B}} & 0 \\ L_{\widehat{E}} & I \end{pmatrix}\begin{pmatrix} D_{\widehat{B}} & 0 \\ 0 & S \end{pmatrix}\begin{pmatrix} U_{\widehat{B}} & U_{\widehat{F}} \\ 0 & I \end{pmatrix}$$
mit $\widehat{B} \approx L_{\widehat{B}} D_{\widehat{B}} U_{\widehat{B}}$. Für die Berechnung des Schur-Komplements S wird aber die **T-Variante** $S = \begin{pmatrix} -L_{\widehat{E}} L_{\widehat{B}}^{-1} & I \end{pmatrix}(\widehat{P}A\widehat{Q}^T)\begin{pmatrix} -U_{\widehat{B}}^{-1}U_{\widehat{F}} \\ I \end{pmatrix}$ verwendet. Nachdem S berechnet wurde, werden $L_{\widehat{E}}$ und $U_{\widehat{F}}$ **verworfen**. Dafür werden (abgesehen von den Permutationen) $L_{\widehat{B}}, D_{\widehat{B}}, U_{\widehat{B}}$ und $\widehat{E}, \widehat{F}$ gespeichert.

3. Setze $A := S$ und wiederhole die beiden ersten Schritte (Multilevel-Strategie).

Am Ende dieses Prozesses erhält man eine Zerlegung $PAQ^T \approx M$ (P und Q sollten nicht verwechselt werden mit denjenigen aus der Gleichung (8.1)), wobei M schematisch wie in Abbildung 8.8 dargestellt wird. Unter Vernachlässigung von Permutationen ist diese Abbildung wie folgt zu verstehen:

- man fängt mit $S_0 = A$ an

[6] aus [17] wird nicht klar, ob am Ende des B-Blocks oder am Ende der "ganzen" Matrix

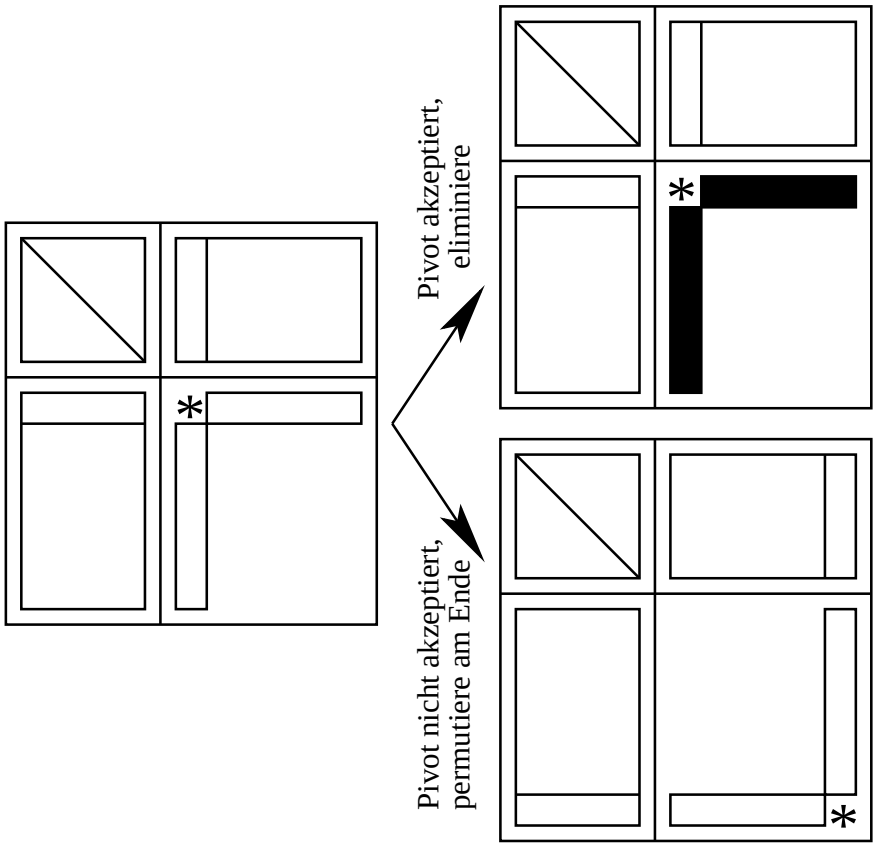


Abbildung 8.7: Pivotisierung in die Diagonale bei Crout-Doolottle

- für $j = 0, 1, \ldots, k-2$, (für $j = k-1$ gilt $S_{k-1} = B_k$), faktorisiere approximativ

$$S_j = \begin{pmatrix} B_{j+1} & F_{j+1} \\ E_{j+1} & C_{j+1} \end{pmatrix} \approx \begin{pmatrix} L_{j+1} & 0 \\ L'_{j+1} & I \end{pmatrix} \begin{pmatrix} D_{j+1} & 0 \\ 0 & S_{j+1} \end{pmatrix} \begin{pmatrix} U_{j+1} & U'_{j+1} \\ 0 & I \end{pmatrix}$$

  mit Schur $S_{j+1} = \begin{pmatrix} -L'_{j+1}L_{j+1}^{-1} & I \end{pmatrix} S_j \begin{pmatrix} -U_{j+1}^{-1}U'_{j+1} \\ I \end{pmatrix}$. Die Blöcke L'_{j+1} und U'_{j+1} werden verworfen.

- Für $j = 1, 2, \ldots, k$, gilt $B_j \approx L_j D_j U_j$. Explizit gespeichert werden nur die Matrizen L_j, D_j, U_j, E_j, F_j. Die Matrizen B_j, C_j, S_j werden nicht explizit gespeichert.

Das Verwerfen der Blöcke L'_{j+1} bzw. U'_{j+1} mag zuerst unverständlich erscheinen. Das verringert den Speicherbedarf erheblich und beim GMRES ersetzt man sie einfach durch $E_{j+1}B_{j+1}^{-1}$ bzw. $B_{j+1}^{-1}F_{j+1}$ (siehe Gleichung (8.2)), obwohl streng gesehen, $E_{j+1}B_{j+1}^{-1}$ bzw. $B_{j+1}^{-1}F_{j+1}$ viel dichter besetzt sind als L'_{j+1} bzw. U'_{j+1}.

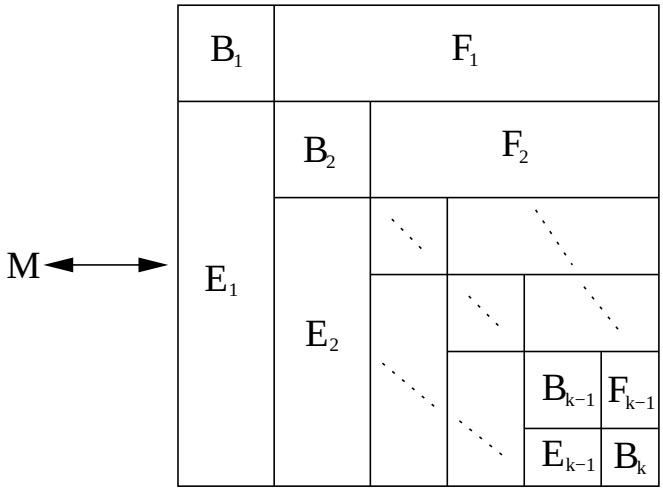


Abbildung 8.8: Implizite Darstellung des Präkonditionierers

ILUPACK benutzt den *GMRES* aus sparskit [56]. Dort verwendet die präkonditionierte GMRES rechte Präkonditionierung (siehe Seite 94). Die Operationen $M \times vector$ und $M^{-1} \times vector$ sollten billig zu bestimmen sein um rechte Präkonditionierung sinnvoll verwenden zu können.

Als Basis für beide Operationen dient folgende approximative Gleichung:

$$S_j = \begin{pmatrix} B_{j+1} & F_{j+1} \\ E_{j+1} & C_{j+1} \end{pmatrix} \approx \begin{pmatrix} I & 0 \\ E_{j+1}B_{j+1}^{-1} & I \end{pmatrix} \begin{pmatrix} B_{j+1} & 0 \\ 0 & S_{j+1} \end{pmatrix} \begin{pmatrix} I & B_{j+1}^{-1}F_{j+1} \\ 0 & I \end{pmatrix} \tag{8.2}$$

Die Operationen $E_{j+1} \times vector$ und $F_{j+1} \times vector$ sind trivial. Approximiert man B_{j+1} durch $L_{j+1}D_{j+1}U_{j+1}$ so sind auch die Operationen $B_{j+1} \times vector$ und $B_{j+1}^{-1} \times vector$ einfach zu bestimmen. Dadurch wird der Operation $S_j \times vector$ rekursiv auf die Operation $S_{j+1} \times vector$ zurückgeführt. So wird $S_0 \times vector$, d.h. $M \times vector$, berechnet.

Aus (8.2) folgt:

$$\begin{aligned} S_j^{-1} \approx \begin{pmatrix} I & B_{j+1}^{-1}F_{j+1} \\ 0 & I \end{pmatrix}^{-1} \begin{pmatrix} B_{j+1} & 0 \\ 0 & S_{j+1} \end{pmatrix}^{-1} \begin{pmatrix} I & 0 \\ E_{j+1}B_{j+1}^{-1} & I \end{pmatrix}^{-1} = \\ = \begin{pmatrix} I & -B_{j+1}^{-1}F_{j+1} \\ 0 & I \end{pmatrix} \begin{pmatrix} B_{j+1}^{-1} & 0 \\ 0 & S_{j+1}^{-1} \end{pmatrix} \begin{pmatrix} I & 0 \\ -E_{j+1}B_{j+1}^{-1} & I \end{pmatrix} \end{aligned} \tag{8.3}$$

Dadurch wird die Operation $S_j^{-1} \times vector$ rekursiv auf die Operation $S_{j+1}^{-1} \times vector$ zurückgeführt. So wird $S_0^{-1} \times vector$, d.h. $M^{-1} \times vector$, berechnet. Aus (8.3) sieht man, dass eine Operation $B_{j+1}^{-1} \times vector$ erspart werden kann.

Ein Aufruf der ILUPACK_V1.1 für die Matrix af23560 sieht so aus:

```
make iluV1.1 condest=1e+1 tau=1.5e-1
```

```
dmainV1.1 1.5e-1 1e+1 20 bolle_input_HB.rua
Matrix: bolle_input_HB.rua: size (23560,23560), nnz=460598(19.6av.)

ILUPACK PARAMETERS:
   droptol= 1.5e-01
   condest= 1.0e+01
   elbow_space_factor=  20
   simple Schur complement
   diagonal entries unmodified
   preproc. initial system: scaling + ddPQ ordering
   reordering subsystems:   scaling + ddPQ ordering
   final subsystem(s) left unchanged
ILUPACK,   multilevel structure
level   1, block size    8746
  local fill-in L   15542(  1.8av),   U   17129(  2.0av),
    sum   32671(  3.7av)
level   1->  2, block size (   8746,  14814)
  local fill-in E  100580( 11.5av pc),F  102814( 11.8av pr),
    sum  203394( 23.3av)
.
.
  local fill-in L      13(  1.0av),   U      15(  1.2av),
    sum      28(  2.2av)
level  26-> 27, block size (     13,     97)
  local fill-in E     327( 25.2av pc),F     337( 25.9av pr),
    sum     664( 51.1av)
level  27, block size      97
switched to full matrix processing
  local fill-in L    4656( 48.0av),   U    4753( 49.0av),
    sum    9409( 97.0av)

total fill-in L&E  313977( 13.3av), U&F  288890( 13.3av),
    sum  626427( 26.6av)

fill-in factor:        1.4
memory usage factor:   1.4
total time:  1.6e+00 [sec]
             1.6e+00 [sec]

refined timings for   ILUPACK multilevel factorization
initial preprocessing:        7.0e-02 [sec]
reorderings remaining levels: 1.5e-01 [sec]
PILUC (sum over all levels):  1.3e+00 [sec]
```

```
ILUTP (if used):              0.0e+00 [sec]
LUPQ (if used):               1.0e-02 [sec]
remaining parts:              4.0e-02 [sec]

  1. right hand side
number of iteration steps: 403
time:  6.5e+00 [sec]
       6.5e+00 [sec]

time matrix-vector multiplication:  3.6e-03 [sec]
residual norms:
initial:  7.6e+00
target:   9.8e-08
current:  4.7e-08
rel. error in the solution:  4.8e-08

  2. right hand side
number of iteration steps: 430
time:  6.9e+00 [sec]
       6.9e+00 [sec]

time matrix-vector multiplication:  3.8e-03 [sec]
residual norms:
initial:  1.0e+05
target:   1.3e-03
current:  6.7e-04
rel. error in the solution:  5.2e-08
```

Wie bereits gesehen, liefert ILUPACK eine unvollständige LDU-Zerlegung $PAQ^T \approx LDU = M$. Die Matrizen L und U werden aber nicht explizit gespeichert. Sie sind implizit in M "enthalten". Dadurch ist es mir nicht klar, was die Zeile
"`fill-in factor:        1.4`"
bei der ILUPACK-Ausgabe bedeutet, zumal die Blöcke L'_j bzw. U'_j durch viel dichtere Blöcke $E_j B_j^{-1}$ bzw. $B_j^{-1} F_j$ approximiert werden.

Wie zu erwarten war, ist die ILU-bestimmende Phase bei der ILUPACK sehr schnell. Das gleiche kann man aber für die GMRES-Phase nicht sagen. Das komplizierte Format, in dem das Ergebnis gespeichert wird, und die nicht ganz einfachen Operationen $M \times vector$ und $M^{-1} \times vector$, verlangsamen die gesamte Laufzeit des iterativen Lösers.

Der Multilevel-Ansatz bei den unvollständigen ILU-Zerlegungen ist eigentlich eine sinnvolle Sache. Dadurch hat man die Möglichkeit, die durch frühere Elimina-

tionsschritte und Dropping verlorengegangenen Eigenschaften, wie zum Beispiel diagonale Dominanz, wieder herzustellen.

Ich glaube, dass ILUPACK intern doppelt verkettete Listenformate verwendet. Dies sind zwar ungeeignet, um auf bestimmte Einträge zuzugreifen, brauchen aber keine zeitkostenden Speicheroperationen (anlegen und freigeben). Durch die Variable `elbow_space_factor` wird ein einmaliger Speicherbereich angelegt und viele Operationen verwenden es (vermute ich wenigstens). Allerdings machen doppelt verkettete Listenformate eine Parallelisierung unmöglich (es entstehen Synchronisationsprobleme).

Die Vor- und Nachteile meines Lösers wurden schon im Abschnitt 8.3 beschrieben.

8.5 Parallelisierungsmöglichkeiten

Bei der "mormalen" multifrontalen Methode gibt es zwei Parallelisierungsansätze ([28],[29]):

- **Tree-Parallelism**:
  Tree-parallelism basiert auf Korollar 4.5. Alle Blätter des (übrig gebliebenen) Baums können unabhängig von einander bearbeitet werden ([55]). Wie Duff et al. ([29]) berichten, ist aus diesem Parallelisierungsansatz nicht viel herauszuholen. Die Anzahl der Blätter nimmt rapide ab, und sehr bald gibt es nichts mehr zu parallelisieren.
  Zusätzlich bringt das inverse based dropping (siehe 6.2.3) ein unerwartetes Hindernis beim Tree-parallelism: Seien z.B. $k_1 < k_2$ zwei parallel zu prozessierende Knoten (d.h. sie liegen in disjunkten Teilbäumen). Seien Q_{k_1} und Q_{k_2} die Mengen aus der Zeile 7 des cond. est. Algorithmus (Seite 110). Alle Knoten aus Q_{k_1} bzw. Q_{k_2} liegen in dem Pfad, welcher k_1 bzw. k_2 mit der Wurzel[7] verbindet (siehe Satz 4.9). Dadurch passiert es oft, dass $Q_{k_1} \cap Q_{k_2} \neq \emptyset$. Also bedarf die Aufdatierung des Vektors v (Zeilen 9-13) entweder einer Synchronisation der Prozesse oder muss von den Rank-1-Updates abgekoppelt und seriell prozessiert werden. Hilfreich ist hier die Tatsache, dass $k_2 \notin Q_{k_1}$ d.h. dropping am Knoten k_2 hat keinerlei v-Aufdatierungen aus den Knoten k_1 nötig. Dadurch kann die v-Aufdatierung am Ende aller Rank-1-Updates seriell durchgeführt werden. Im Endeffekt rechtfertigt der geringe Gewinn nicht die hohen Kosten. Daher habe ich diesen Ansatz nicht weiter verfolgt.

- **Node-Parallelism**:
  Node-Parallelism ist die parallele Durchführung des Rank-1-Updates (bei je-

[7] des Eliminationsbaums

dem Knoten). Die Zeilenmenge (oder Spaltenmenge, falls FORTRAN) wird partitioniert, und das Rank-1-Update wird durch den für die Zeile verantwortlichen Prozessor errechnet. Dieser Parallelisierungsansatz ist einfach zu implementieren und Duff et al. berichten, dass er gute speed-ups bringt.

Wie schon erwähnt speichert die mf_incomplete-Variante die frontalen Matrizen zeilenweise, und es werden keinerlei Informationen über Spalten explizit gespeichert. Bei der not_mf-Variante wird die Matrix zwar auch zeilenweise gespeichert, aber für jede Spalte gibt es eine (explizit gespeicherte) doppelt verkettete Liste, um sie schnell zu generieren, wenn es notwendig ist. Wenn beim Rank-1-Update ein neuer Eintrag entsteht, z.B. in einer Koordinaten (i, j), die bisher mit Null besetzt war, muss man außer in die Datenstruktur[8] der i-ten Zeile diesen Eintrag auch in die Liste der j-ten Spalte einfügen. Dadurch kann es vorkommen, dass verschiedene Prozesse gleichzeitig auf dieselbe (Spalten)Liste zugreifen. Dieses Synchronisationsproblem ist das gleiche wie das gerade besprochene beim Tree-Parallelism. Mir ist nicht bekannt, wie die Synchronisation der einzelnen Elemente eines Arrays der Länge $\mathcal{O}(n)$ billig zu schaffen ist. Die Herangehensweise, bei der die Threads eine Sperre über das Array "verhängen", bringt große Verzögerungen mit sich (es entsteht ein Engpass). Für den Node-Parallelism scheint nur die mf_incomplete-Variante geeignet zu sein, obwohl, seriell gesehen, die mf_incomplete-Variante der not_mf-Variante unterlegen ist.

Zur Bewertung der Qualität eines parallelen Algorithmus wird der so gennante *Speed-up* verwendet. Der wird durch:

$$S(p) = \frac{\text{Ausführungszeit des seriellen Algorithmus}}{\text{Ausführungszeit der parallelen Algorithmus auf } p \text{ Prozessoren}}$$

definiert. Falls es einen $\lambda > 0$ und ein $p_0 > 0$ gibt, so dass

$$\forall p > p_0,\ \frac{S(p)}{p} \geq \lambda$$

gilt, dann bezeichnen wir den parallelen Algorithmus als *gut skalierbar*, ansonsten *schlecht skalierbar*.

Um Programme zu parallelisieren, ist es nötig einen Parallelrechner zu haben. Solche Rechner werden anhand des Speichers in zwei Kategorien klassifiziert. In der ersten Kategorie verfügt jeder Prozessor über seinen eigenen lokalen Speicher, ein gemeinsamer Speicher steht nicht zur Verfügung. Diese Rechner bezeichnet man auch als Rechner mit verteiltem Speicher (sog. distributed memory). In den zweiten Kategorie teilen sich alle Prozessoren einen zentralen gemeinsamen Speicher (sog. shared memory), auf den alle Prozessoren gleichberechtigt zugreifen.

[8] in meinen Fall, AVL-Baum

Müssen Prozessoren bei der Bearbeitung ihrer Teilprobleme Daten austauschen, so findet die notwendige Kommunikation über den gemeinsamen Speicher statt.

Für Rechner der ersten Kategorie erfolgt die Kommunikation normalerweise über Netzwerkschnittstellen. Da es in der Regel sehr aufwändig ist, den Prozessoren paarweise einen direkten Kommunikationskanal zu ermöglichen, werden andere Verbindungsmuster, sog. Topologien, verwendet. Die Prozessoren werden z.B. in einem Gitter, einem Torus, einem Hypercube oder einem Ring angeordnet. Die Kommunikation zwischen zwei nicht benachbarten Prozessoren verläuft dann über die dazwischenliegenden Prozessoren. MPI (Message Passing Interface, [3]) spezifiziert eine Sammlung von Routinen und Umgebungsvariablen, welche die Parallelisierung auf Rechnern mit verteiltem Speicher für die Programmiersprachen C, C++ und FORTRAN standardisiert. Bei jedem Versenden wird eine Nachricht mit einem *tag* versehen, zur Unterscheidung verschiedener Nachrichten. Ein Kommunikationsvorgang wird durch ein Tripel *(Sender,Empfänger,tag)* eindeutig beschrieben. Typischerweise enthält eine Nachricht ein Feld aus built-in-Datentypen, z.B. integer's oder double's.

Parallelisierung auf Rechnern mit verteiltem Speicher ist nicht geeignet für Programme die viel Kommunkation benötigen. Beim Gauß-Algorithmus ist das der Fall: Für jedes $k = 1, 2, \ldots, n-1$, von einem Master-Prozessor aus, werden die Daten an jeden Slave-Prozessor geschickt, prozessiert und anschliessend an den Master zurückgeschickt. Hier wird die Kommunikation überwiegen. Obwohl MPI meine bevorzugte Parallelisierungsmöglichkeit ist, müsste ich einsehen, dass es für meinen Fall ungeeignet war. Die Parallelisierungstests, die ich mit MPI durchführte, brachten nur Verlangsamung statt speed-up.

Für Rechner der zweiten Kategorie erfolgt die Kommunikation über den gemeinsamen Speicher. Ein wesentlicher Punkt bei der Parallelisierung mit gemeinsamem Speicher ist die Tatsache, dass Speicherzugriffe auf globale Daten mit Hilfe von Sperrmechanismen synchronisiert werden müssen, um nicht z.B. bei gleichzeitigem Schreibzugriff (sog. *Race Conditions*) einen inkonsistenten Speicherinhalt zu erhalten. OpenMP API (Open Multi Processing Application Program Interface, [4]) spezifiziert eine Sammlung von Compiler-Direktiven, Bibliotheken und Umgebungsvariablen, welche die Parallelisierung auf Rechnern mit gemeinsamen Speicher für die Programmiersprachen C, C++ und FORTRAN standardisiert. Die Parallelisierung erfolgt hier auf Thread-Ebene. Alle OpenMP-Direktiven für C++ werden über `#pragma`-Präprozessor-Direktiven spezifiziert. Jede dieser Direktiven beginnt standardmäßig mit `#pragma omp`. Es folgt die Beschreibung einiger wichtiger OpenMP-Compiler-Direktiven[9].

- `Parallele Region / #pragma omp parallel`

[9] Dies ist eine oberflächliche Beschreibung. Für eine vollständige Beschreibung siehe [4]

Diese OpenMP-Direktive ist das wesentliche Konstrukt, welches eine so genannte *parallele Region* startet. Die Syntax sieht wie folgt aus:

```
#pragma omp parallel if(scalar-expr.) num_threads(integer-expr.)
private(variable-list) shared(variable-list)
{
   <Anweisungsblock>
}
```

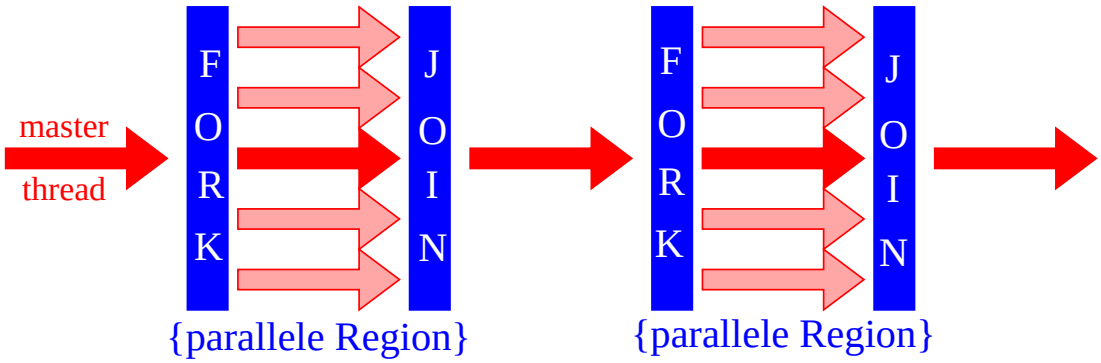


Abbildung 8.9: Das Fork-Join-Prinzip beim OpenMP: Der Master Thread erzeugt zu Beginn der parallelen Region ein Thread-Team. Am Ende werden alle Threads, bis auf den Master Thread, beendet.

Die Abarbeitung des Quellcodes eines beliebigen OpenMP-Programmes beginnt mit einem einzigen Thread, dem so genannten Master Thread. Sobald eine parallele Region erreicht wird und das `scalar-expression` bei der `if`-Klausel als wahr bewertet wird, erzeugt dieser ein ganzes Team von Threads, deren Anzahl entweder über die Klausel `num_threads(integer-expression)` oder über die Umgebugsvariable `OMP_NUM_THREADS` gesteuert wird. Über die Klausel `private(variable-list)` wird eine Liste von Variablen vereinbart, die *privat* sind, d.h. für jede Variable der Liste eine *uninitialisierte* Kopie für jeden Thread des Teams angelegt wird. Gemeinsame Variablen, die im gemeinsamen Adressraum liegen und damit von jedem Thread verändert werden können[10], werden über die Klausel `shared(variable-list)` vereinbart. Jede Anweisung aus dem Anweisungsblock wird von allen Teammitglidern parallel ausgeführt, sofern dies nicht durch spezielle OpenMP-Direktiven "verhindert" wird. Am Ende der parallelen Region werden alle Threads durch eine implizite Barriere synchronisiert. Anschließend fährt ausschließlich der Master Thread weiter (Abbildung 8.9).

- `Kritische Bereiche / #pragma omp critical`

Durch die OpenMP-Direktive

[10] Veränderung sollten mittels `critical`- oder `single`-Klausel implementiert werden

```
#pragma omp critical
{
   <Anweisungsblock>
}
```

werden so genannte *kritische Bereiche in parallelen Regionen* definiert. Diese Klausel bewirkt, dass sich nur jeweils ein Thread des Teams in diesem kritischen Bereich befinden darf, um den entsprechenden Anweisungsblock auszuführen.

- `Parallele Schleifen / #pragma omp for`

Wir haben gesehen, dass jede Anweisung des Anweisungsblocks innerhalb einer parallelen Region von allen Threads ausgeführt wird. OpenMP stellt allerdings auch Direktiven zur Verfügung, die die Arbeit auf einzelne Threads verteilt. Die wohl wichtigste Möglichkeit dieser Art ist die Parallelisierung von Schleifen.

```
#pragma omp for if(scalar-expr.) num_threads(integer-expr.)
private(variable-list) shared(variable-list)
schedule(kind,chunk_size)
{
   for( i=a; i<e; i++)
   {
      <Schleifenrumpf>
   }
}
```

Die Klauseln `if`, `num_threads`, `private` und `shared` sind identisch mit denen aus der `#pragma omp parallel`-Direktive. Die Schleifenvariable `i` und die Ausdrücke `a` und `e` stehen für Integer-Variablen. Diese dürfen innerhalb des Schleifenrumpfes nicht verändert werden. Die Schleifenvariable `i` wird implizit zu einer privaten Variable für jeden Thread des Teams gemacht, damit die einzelnen Iterationen völlig unabhängig voneinander und auch in einer eventuell anderen Reihenfolge durch die Threads abgearbeitet werden können. Die Aufteilung der Iterationen (sog. *schedule*) auf die einzelnen Threads erfolgt über die Wahl von `kind`:

static Die Iterationen werden zunächst in Blöcke der Größe `chunk_size` aufgeteilt. Diese werden anschließend statisch auf die einzelnen Threads verteilt, und zwar zyklisch.

dynamic Die Iterationen werden in Blöcken der Größe `chunk_size` den verschiedenen Threads zugewiesen, wenn diese durch die Threads angefordert werden. Ein Thread führt zunächst seinen Iterationsblock aus und fordert dann einen neuen Block an, solange bis alle Blöcke abgearbeitet sind.

guided Die Threads fangen mit Blöcken der Größe `chunk_size` an. Wenn ein Thread mit seinem Block fertig wird, wird ihm der nächste Block dyna-

misch zugewiesen. Im Unterschied zu **dynamic**, nehmen hier die Blöckgrößen annähernd exponenziell ab.

runtime Hier wird die Entscheidung über die Aufteilung der Iterationen bis zur Laufzeit des Programms verzögert und dann über die Umgebugsvariable `OMP_SCHEDULE` ermittelt.

- `#pragma omp single`

In einer parallelen Region, stellt die OpenMP-Direktive

```
#pragma omp single
{
   <Anweisungsblock>
}
```

sicher, dass der entsprechende Anweisungsblock nur von einem Thread, und von keinen anderen, ausgeführt wird. Dies ist z.B. sinnvoll, wenn globale Daten initialisiert werden sollen.

OpenMP Parallelisierung für die mf_incomplete-Variante macht Sinn und ist einfach zu implementieren. Alles, was man tun müss, ist die for-Schleife, in der das Rank-1-Update stattfindet, mittels:

```
#pragma omp parallel for if(..) num_threads(..) schedule(static) ...
```

zu parallelisieren. Bei der Klausel `schedule` ist nicht offensichtlich, welche der zur Verfügung stehenden Möglichkeiten `static, dynamic, guided` oder `runtime` auszuwählen ist. Nach umfangreichen Tests stellte sich heraus, dass `static` die beste Auswahl ist.

Die Erzeugung der Threads kostet Ressourcen und Zeit. Daher überprüfe ich erst, wieviel Arbeit es zu verrichten gibt (bei der Klausel `if(..)`), und dann bestimme ich die Anzahl der Threads (bei der Klausel `num_threads(..)`), die die Arbeit erledigen sollen. Das bedeutet aber auch, dass je mehr Prozessoren das Programm nutzen darf, desto seltener **alle gleichzeitig** beansprucht werden (es gibt halt nicht immer genügend viel Arbeit). Die Folge ist eine Abnahme des Speed-ups für hohe Prozessorenzahlen (diesbezüglich siehe auch Abbildungen 8.10, 8.11 und 8.12). Die Skalierbarkeit ist also deutlich eingeschränkt.

Würde man andereseits immer die gleiche Threadanzahl benutzen, dann wäre der Speed-up gering.

Die Parallelisierung von ILU's enthält etwas Widersprüchliches in sich. Je besser das Dropping wirkt, desto weniger zu parallelisierende Arbeit gibt es, und

dadurch verringert sich das Speed-up. Je schlechter das Dropping wirkt, desto mehr zu parallelisierende Arbeit gibt es und dadurch erhöht sich das Speed-up. Guter Speed-up für mehrere Prozessoren könnte somit ein Indiz für großes Fill-in sein.

Folgende Plots veranschaulichen das Speed-up der die ILU-bestimmenden Hauptroutine[11]. Testmaschine war "`wmai30.math.uni-wuppertal.de`", eine shared-memory Maschine an der Universität Wuppertal, mit den Parametern: Sun Ultra Sparc III 1.2GHz, 8 Prozessoren mit je 8MB L2 Cache, Betriebsystem Solaris 9, SunOS 5.9. Kompiliert wurde mit den Sun CC Compiler, unter Verwendung von SunStudio9, mit den `-fast` und `-xopenmp` Optionen.

Das überlineare Speed-up für kleine Prozessorzahlen könnte eine Folge von Cache-Effekten sein. Bekanntermaßen ist ein Level-2-Cache-Zugriff ungefähr fünf Mal schneller als ein Hauptspeicherzugriff (RAM).

[11] also nicht des ganzen Programms

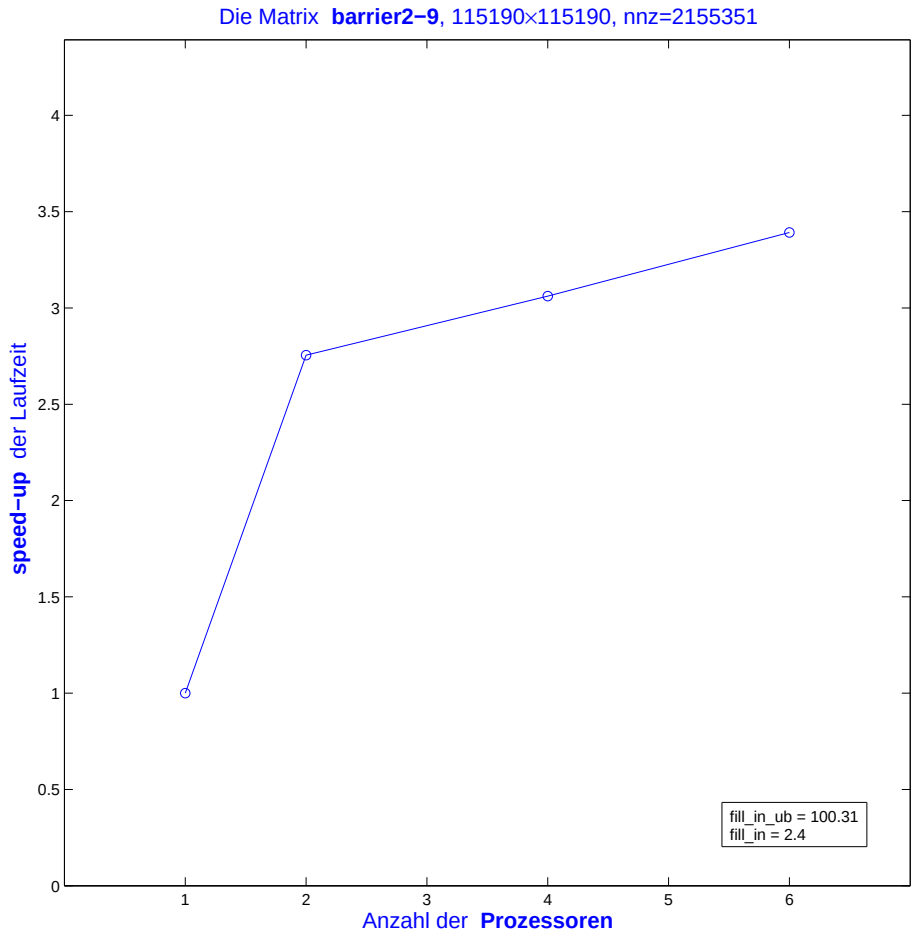


Abbildung 8.10:

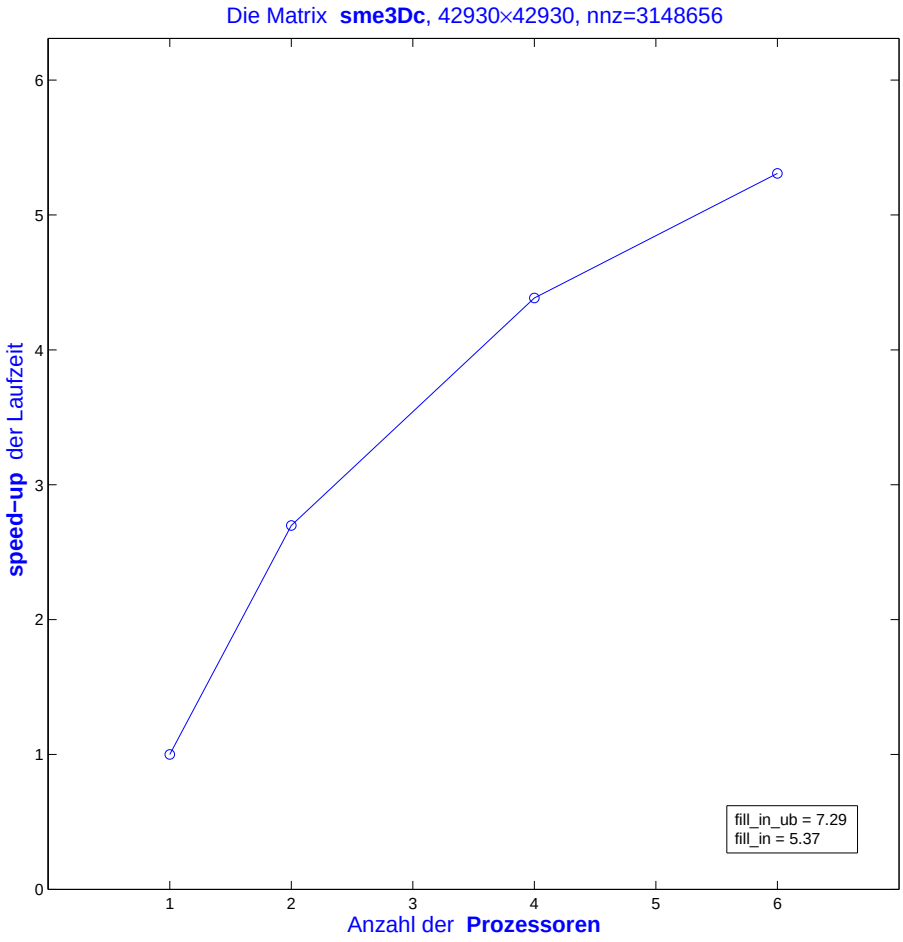


Abbildung 8.11:

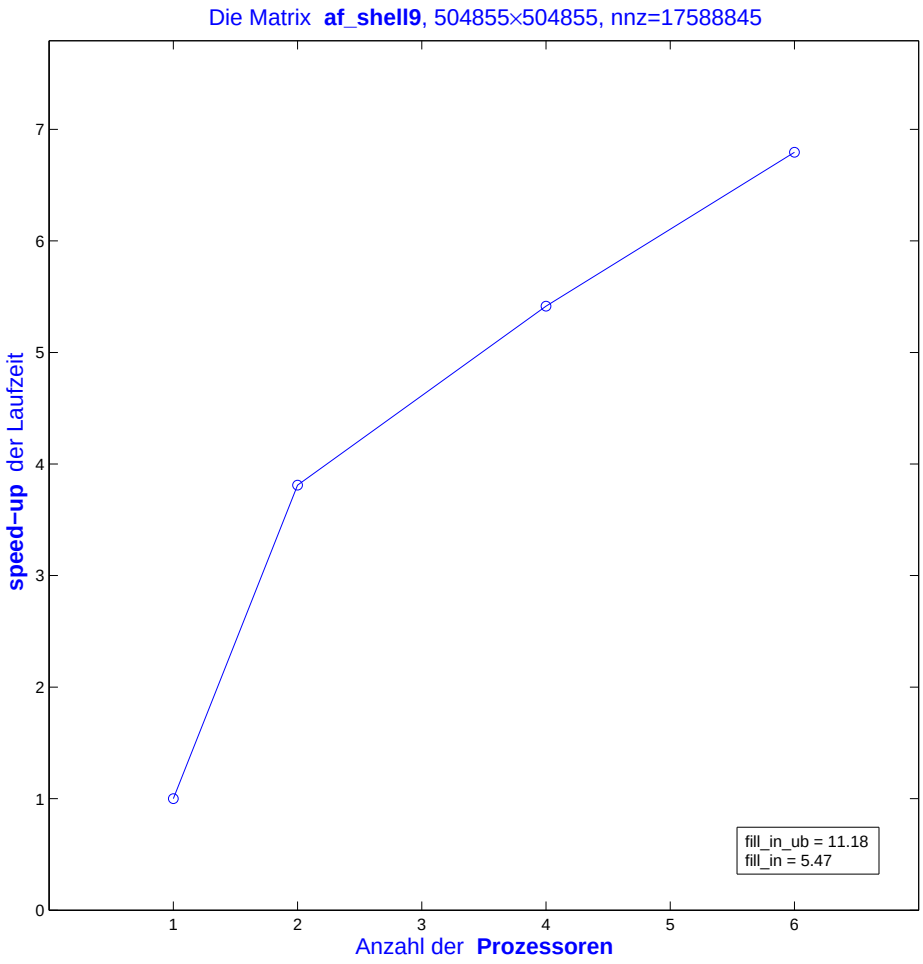


Abbildung 8.12:

Danksagung

Ein langer Weg hat seinen Abschluß gefunden. Nun ist es Zeit, innezuhalten und zurückzuschauen, um sich der beschrittenen Pfade und der treuen Begleiter zu erinnern.

Im Nachfolgenden möchte ich mich bei Personen bedanken, die mir auf vielfältige Art und Weise geholfen haben und so diese Arbeit ermöglichten.

Mein herzlichster Dank gilt:

- Meinem Doktorvater Prof. Dr. Andreas Frommer für die kompetente Betreuung meiner Arbeit, für gute Ideen und Anregungen. Er hat mir immer Freiraum für meinen eigenen Weg gegeben,
- Prof. Dr. Bruno Lang für seine Hilfe und die Übernahme des Korreferates,
- Holger Arndt für die unermüdliche Erklärung meiner vielen Fragen,
- Meinen Kollegen, insbesondere Peter Langer, Thomas Beelitz und Karsten Blankenagel, für das angenehme Arbeitsklima,
- Frau Brigitte Schultz für ihre immer bedingungslose Hilfbereitschaft,
- Meiner Frau Elona, meinen Eltern und meinen Schwiegereltern die mir mit viel Liebe und großem Verständnis den Rückhalt für meine Arbeit gegeben haben,
- Iain Duff für die Kopie des MC64-Codes,
- Timothy Davis für den frei erhältlichen AMD-Code und der University of Florida Sparse Matrix Collection,
- Mathias Bollhöfer und Yousef Saad für den frei erhältlichen ILUPACK-Code.

Literaturverzeichnis

[1] *BLAS Documentation*. `http://www.netlib.org/blas/blast-forum/`.

[2] *Matrix Market*. `http://math.nist.gov/MatrixMarket`, National Institute of Standards.

[3] *MPI Documentation*. `http://www.mpi-forum.org/docs/docs.html`.

[4] *OMP Documentation*. `http://www.openmp.org/drupal/node/view/8/`.

[5] A.V. Aho, M.R. Garey, and J.D. Ullman. The transitive reduction of a directed graph. *SIAM J. Comput.*, 1(2):131–137, June 1972.

[6] P.R. Amestoy, T.A. Davis, and I.S. Duff. An approximate minimum degree ordering algorithm. *SIAM J. Matrix Anal. Appl.*, 17(4):886–905, October 1996.

[7] C. Ashcraft, S. C. Eisenstat, and R.F. Lucas. personal communication.

[8] M. Benzi, C.D. Meyer, and M. Tuma. A sparse approximate inverse preconditioner for the conjugate gradient method. *SIAM J. Sci. Comput.*, 17(5):1135–1149, September 1996.

[9] M. Benzi and M. Tuma. A sparse approximate inverse preconditioner for nonsymmetric linear systems. *SIAM J. Sci. Comput.*, 19(3):968–994, May 1998.

[10] M. Benzi and M. Tuma. Orderings for factorized sparse approximate inverse preconditioners. *SIAM J. Sci. Comput.*, 21(5):1851–1868, 2000.

[11] M. Bollhöfer. A robust ILU with pivoting based on monitoring the growth of the inverse factors. *Linear Algebra Appl.*, 338(1–3):201–218, 2001.

[12] M. Bollhöfer. A robust and efficient ILU that incorporates the growth of the inverse triangular factors. *SIAM J. Sci. Comput.*, 25(1):86–103, 2003.

[13] M. Bollhöfer and Y. Saad. ILUPACK V1.1-preconditioning software package, released on december 3, 2004. Available at `http://www.math.tu-berlin.de/ilupack/`.

[14] M. Bollhöfer and Y. Saad. ILUs and Factorized Approximate Inverses are strongly related. Part ii: Applications to stabilization. Technical Report umsi-2000-70, University of Minnesota at Minneapolis, Dep. of Computer Science and Engineering, 2000.

[15] M. Bollhöfer and Y. Saad. ILUs and Factorized Approximate Inverses are strongly selated. Part i: Overview of results. Technical Report umsi-2000-39, Minnesota Supercomputer Institute, University of Minnesota, 2000.

[16] M. Bollhöfer and Y. Saad. On the relations between ILUs and factored approximate inverses. *SIAM J. Matrix Anal. Appl.*, 24(1):219–237, 2002.

[17] M. Bollhöfer and Y. Saad. Multilevel preconditioners constructed from inverse-based ILUs. *SIAM J. Sci. Comput.*, 27(5):1627–1650, 2006.

[18] J.R. Bunch and L. Kaufman. Some stable methods for calculating inertia and solving symmetric linear equations. *Math. Comp.*, 31:163–179, 1977.

[19] G. Carpaneto and P. Toth. Solution of the assignment problem. *ACM Trans. Math. Software*, pages 104–111, 1980.

[20] A.K. Cline, C.B. Moler, G.W. Stewart, and J.H. Wilkinson. An estimate for the condition number of a matrix. *SIAM J. Num. Anal.*, 16(2):368–375, 1979.

[21] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to algorithms, 2nd edition.* The MIT Press, Cambridge, Massachusetts 02142, 2001.

[22] T.A. Davis. *UMFPACK Version 5.0.* `http://www.cise.ufl.edu/research/sparse/umfpack/`.

[23] T.A. Davis. *University of Florida Sparse Matrix Collection.* `http://www.cise.ufl.edu/research/sparse/matrices/`.

[24] T.A. Davis, J.R. Gilbert, S.I. Larimore, and E.G. Ng. Colamd, a column approximate minimum degree ordering algorithm. *ACM Trans. Math. Software*, 30(3):377–380, September 2004.

[25] I.S. Duff. *HSL - Harwell Subroutine Library.* `http://www.cse.clrc.ac.uk/nag/hsl/`.

[26] I.S. Duff. Algorithm 575: Permutations for a zero-free diagonal. *ACM Trans. Math. Software*, 7:387–390, 1981.

[27] I.S. Duff. On algorithms for obtaining a maximal transversal. *ACM Trans. Math. Software*, 7:315–330, 1981.

[28] I.S. Duff. Parallel implementation of multifrontal schemes. *Parallel Computing*, 3(3):193–204, July 1986.

[29] I.S. Duff, P.R. Amestoy, and J.Y. L'Excellent. Multifrontal parallel distributed symmetric and unsymmetric solvers. Technical Report RAL-TR-1998-051, Rutherford Appleton Laboratory, Oxon OX11 0QX, July 1998.

[30] I.S. Duff, A.M. Erisman, and J.K. Reid. *Direct Methods for Sparse Matrices.* Oxford University Press, New York, 1986.

[31] I.S. Duff and J. Koster. The design and use of algorithms for permuting large entries to the diagonal of sparse matrices. *SIAM J. Matrix Anal. Appl.*, 20(4):889–901, 1999.

[32] I.S. Duff and J. Koster. On algorithms for permuting large entries to the diagonal of a sparse matrix. *SIAM J. Matrix Anal. Appl.*, 22(4):973–996, 2001.

[33] I.S. Duff and J.K. Reid. The multifrontal solution of indefinite sparse symmetric linear equations. *ACM Trans. Math. Software*, 9(3):302–325, September 1983.

[34] L. Fox. *An introduction to numerical linear algebra.* Oxford University Press, 1964.

[35] A. Frommer. Algorithmen auf Graphen und dünn besetzte Matrizen, 2004.

[36] A. George and J.W.H. Liu. The evolution of the minimum degree ordering algorithm. *SIAM Review*, 31(1):1–19, March 1989.

[37] J.R. Gilbert. Predicting structure in sparse matrix computations. *SIAM J. Matrix Anal. Appl.*, 15(1):62–79, 1994.

[38] J.R. Gilbert, C. Moler, and R. Schreiber. Sparse matrices in MATLAB: design and implementation. *SIAM J. Matrix Anal. Appl.*, 13(1):333–356, 1992.

[39] J.R. Gilbert, E.G. Ng, and B.W Peyton. An efficient algorithm to compute row and column counts for sparse Cholesky factorization. *SIAM Journal on Matrix Analysis and Applications*, 15(4):1075–1091, 1994.

[40] J.R. Gilbert and T. Peierls. Sparse partial pivoting in time proportional to arithmetic operations. *SIAM J. Sci. Stat. Comput.*, 9(5):862–874, September 1988.

[41] G.H. Golub and C. Van Loan. *Matrix computations, 3rd Edition*. The John Hopkins University Press, Baltimore, 1996.

[42] J.E. Hopcroft and R.M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 4(2):225–231, 1973.

[43] The MathWorks Inc. *MATLAB Version 6.1 Release 12*. Natic, MA, 2001.

[44] J.W.H. Liu. Modification of the minimum-degree algorithm by multiple elimination. *ACM Trans. Math. Software*, 11(2):141–153, June 1985.

[45] J.W.H. Liu. A tree model for sparse symmetric indefinite matrix factorisation. *SIAM J. Matrix Anal. Appl.*, 9(1):26–39, January 1988.

[46] J.W.H. Liu. The role of elimination trees in sparse factorisation. *SIAM J. Matrix Anal. Appl.*, 11(1):134–172, January 1990.

[47] J.W.H. Liu. The multifrontal method for sparse matrix solution: Theory and practice. *SIAM Review*, 34(1):82–109, March 1992.

[48] J.W.H. Liu and J.A. George. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[49] J.W.H. Liu, E.G. Ng, and B.W. Peyton. On finding supernodes for sparse matrix computations. *SIAM J. Matrix Anal. Appl.*, 14(1):242–252, January 1993.

[50] J. Meijerink and H.A. van der Vorst. An iterative solution method for linear systems of which the coefficient matrix is a symmetric M-matrix. *Math. Comp.*, 31:148–162, 1977.

[51] M Olschowka and A. Neumeier. A new pivoting strategy for gaussian elimination. *Linear Algebra Appl.*, 240:131–151, 1996.

[52] T. Ottmann and P. Widmayer. *Algorithmen und Datenstrukturen, 4. Auflage*. Spektrum Akademischer Verlag, 2002.

[53] C. Paige and M. Saunders. Solution of sparse indefinite systems of linear equations. *SIAM J. Numer. Anal.*, 12:617–629, 1975.

[54] S. Pissanetsky. *Sparse Matrix Technology*. Academic Press, London, 1984.

[55] A. Pothen and C. Sun. A mapping algorithm for parallel sparse cholesky factorisation. *SIAM J. Sci. Comput.*, 14(5):1253–1257, September 1993.

[56] Y. Saad. SPARSKIT: a basic toolkit for sparse matrix computation. Available at `http://www-users.cs.umn.edu/~saad/software/SPARSKIT/sparskit.html`.

[57] Y. Saad. ILUT: A dual threshold incomplete ILU preconditioner. *Numerical Linear Algebra with Applications*, 1(4):387–402, 1994.

[58] Y. Saad. *Iterative methods for sparse linear systems, 2nd edition*. SIAM, Philadelphia, PA, 2003.

[59] Y. Saad. Multilevel ILU with reorderings for diagonal dominance. *SIAM J. Sci. Comput.*, 27(3):1032–1057, 2005.

[60] Y. Saad, N. Li, and E. Chow. Crout versions of ILU for general matrices. *SIAM J. Sci. Comput.*, 25(2):716–728, 2003.

[61] Y. Saad and M. H. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on Matrix Analysis and Applications*, 7(3):856–869, 1986.

[62] Y. Saad and B. Suchomel. ARMS: an algebraic recursive multilevel solver for general sparse linear systems. *Linear Algebra Appl.*, 9(5):359–378, July/August 2002.

[63] P. Sonneveld. CGS, a fast Lanczos-type solver for nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 10(1):36–52, 1989.

[64] R.E. Tarjan. *Data structures and network Algorithms*. SIAM, Philadelphia, Pennsylvania, 1983.

[65] M. Tismenetsky. A new preconditioning technique for solving large sparse linear systems. *Linear Algebra Appl.*, 154–156:331–353, 1991.

[66] H. van der Vorst. BiCGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 13(2):631–644, 1992.

[67] M. Yannakakis. Computing the minimum fill-in is NP-complete. *SIAM J. on Algebraic and Discrete Methods*, 2(1):77–79, 1981.

Index