Algorithmen auf Graphen und dünn besetzte Matrizen

Prof. Dr. Andreas Frommer

Wintersemester 2003/04

Bergische Universität Wuppertal
Fachbereich Mathematik

Inhaltsverzeichnis

Literatur

[1] Duff, I., Erisman, A., Reid, J.; Direct Methods for Sparse Matrices, Oxford University Press (1986)

[2] George, A., Liu, J.: Computer Simulation of large Sparse Positive Definite Systems, Prentice Hall (1981)

[3] Pissanetzki, S.: Sparse Matrix Technology, Academic Press (1984)

Einleitung

Definition Eine Matrix $A \in \mathbb{R}^{n \times n}$ heißt *dünn besetzt*, wenn so viele Einträge Null sind, dass es sich lohnt, dies in Algorithmen zur Lösung von $Ax = b$ auszunutzen. Andernfalls heißt A *dicht* oder *voll besetzt*.

Beispiel Diskretisierung partieller Differentialgleichungen führt auf Matrizen, bei denen pro Zeile eine feste Zahl (z.B. 5) von Einträgen $\neq 0$ ist.
Fast alle „großen“, in der Natur vorkommenden Matrizen sind dünn besetzt.

Inhalte dieser Vorlesung sind „Zutaten“ für die Gauß-Elimination für dünn besetzte Matrizen, insbesondere

- Fragestellungen aus Graphentheorie/diskrete Mathematik bei der Bestimmung von Permutationen.
- Fragestellungen aus numerischer Analysis bei der Beurteilung von Rundungsfehlern.
- Fragestellungen aus der praktischen Informatik zur Verwendung geeigneter Datenstrukturen und zur Aufwandsanalyse.

KAPITEL 1

Dünn besetzte Matrizen: Datenstrukturen und elementare Operationen

Abschnitt 1.1

Dünn besetzte Vektoren

Bei einem Vektor $x \in \mathbb{R}^n$, dessen Einträge überwiegend 0 sind, lohnt es nicht, einen Speicherplatz $\mathcal{O}(n)$ zur Verfügung zu stellen.

1.1.1 Definition

a) Für $x \in \mathbb{R}^n$ ist $struct(x) = \{i \in \{1, \ldots, n\} : x_i \neq 0\}$ und $nnz(x) = |struct(x)|$.

   (nnz = „number of non zeros“ siehe MATLAB)

b) Für $x \in \mathbb{R}^n$ ist das *gepackte Format* ein Paar $(ind, va\ell)$ mit $ind \in \mathbb{N}^{nnz(x)}$, $va\ell \in \mathbb{R}^{nnz(x)}$ und mit

$$va\ell(i) = x_{ind(i)}, \ i = 1, \ldots, nnz(x).$$

   (Wir schreiben () in ind und $va\ell$ statt untere Indizes, um Komponenten zu bezeichnen.)

1.1.2 Beispiel

$x = (0, 0.1, 2.3, 0, 0, 4.5, 0, 0.6, 7.8, 0) \in \mathbb{R}^{10}$

i	1	2	3	4	5
ind	2	3	6	8	9
$va\ell$	0.1	2.3	4.5	0.6	7.8

oder auch

i	1	2	3	4	5
ind	6	9	2	8	3
$va\ell$	4.5	7.8	0.1	0.6	2.3

$struct(x)$ ist mit ind identifizierbar.

1.1.3 Algorithmus
$gather(x, n, ind, va\ell, nnz)$

{Berechnet das gepackte Format $(ind, va\ell)$ eines als volles Feld gegebenen Vektors $x \in \mathbb{R}^n$}

```
j := 0
for i = 1 to n do
  if x_i ≠ 0 then
    j++
    ind(j) := i
    vaℓ(j) := x_i
  end if
end for
nnz := j
```

Aufwand: $\mathcal{O}(n)$.

1.1.4 Algorithmus
$scatter(x, n, ind, va\ell, nnz)$

{Bestimmt x als Feld der Länge n aus dem gepackten Format $(ind, va\ell)$ und n}

siehe Übungsblatt 1

Aufwand: $\mathcal{O}(n)$, falls x explizit auf Null initialisiert werden muss, andernfalls $\mathcal{O}(nnz)$.

Wir betrachten nun die Operation
(1.1.1)
$y := \alpha x + y$ „SAXPY“-Operation (**a**lpha **x** **p**lus **y**) , $\alpha \in \mathbb{R}, \alpha \neq 0$.

1.1.5 Lemma
Sei $z = \alpha x + y,\ x, y, z \in \mathbb{R}^n, \alpha \neq 0$. Dann gilt

$$\begin{aligned} struct(z) &= struct(x) \cup struct(y) \\ nnz(z) &= nnz(x) + nnz(y) - |struct(x) \cap struct(y)|. \end{aligned}$$

Beweis: Trivial, aber das Lemma ist falsch! *Erklärung*: Wir nennen eine *zufällige Null* in z einen Eintrag $z_i = 0$ mit $y_i \neq 0$. Das Lemma ist nur korrekt, wenn keine zufälligen Nullen auftreten, was wir ab jetzt voraussetzen. □

Trivial-Algorithmus für SAXPY

scatter x
scatter y
$y = \alpha x + y$ mit vollen Vektoren ($\rightsquigarrow \mathcal{O}(n)$)
gather y

Wir suchen eine Variante, die nur einen vollen Vektor benötigt und Aufwand $\mathcal{O}(nnz(x) + nnz(y))$ besitzt.

Idee:

scatter x (nach w)
berechne $w_i = \alpha x_i + y_i$ für $i \in struct(x) \cap struct(y)$ und speichere in y (Lauf über y)
berechne $w_i = \alpha x_i + y_i$ für $i \in struct(x) \setminus struct(y)$ und speichere in y (Lauf über x)
(die Werte y_i für $i \in struct(y) \setminus struct(x)$ bleiben unverändert.)

1.1.6 Algorithmus (a)

$SAXPY(n, ind_x, ind_y, val_x, val_y, nnz(x), nnz(y), \alpha)$
{Berechnet bei gepackten Vektoren x, y ein SAXPY $y := \alpha x + y$. Der volle Hilfsvektor $w \in \mathbb{R}^n$ ist auf Null initialisiert und wird auch so wieder zurückgeliefert.}

scatter $(w, n, ind_x, val_x, nnz(x))$ {w enthält x}
for $i = 1$ to $nnz(y)$ **do** {Lauf über y}
  if $w_{ind_y(i)} \neq 0$ **then**
    $val_y(i) := \alpha w_{ind_y(i)} + val_y(i)$
    $w_{ind_y(i)} := 0$
  end if
end for
$j := nnz(y)$
for $i = 1$ to $nnz(x)$ **do** {Lauf über x}
  if $w_{ind_x(i)} \neq 0$ **then**
    $j{+}{+}$
    $ind_y(j) := ind_x(i)$
    $val_y(j) := \alpha w_{ind_x(i)}$
    $w_{ind_x(i)} := 0$
  end if
end for
$nnz(y) := j$

Beachte: y ist im Allgemeinen nach Ende von Algorithmus 1.1.6 nicht (mehr) geordnet.
Aufwand: $\mathcal{O}(nnz(x) + nnz(y))$

Mögliche Modifikation von Algorithmus 1.1.6:

a) Es wird erreicht, dass w das Ergebnis $\alpha x + y$ als voll besetzter Vektor darstellt. Dazu wird anfangs das *scatter* mit y statt mit x durchgeführt. Berechnet man ein GAXPY (**G**eneral SAXPY)

$$y := \sum_{j=1}^{m} \alpha_j x^j \; + \; y \qquad \text{Folge von SAXPY's}$$

als eine Folge von SAXPYs, so kann bei dieser Variante dann ab dem zweiten SAXPY das *scatter* jeweils entfallen; das 'Einsammeln' in den dünn besetzten Vektor y ist nur beim letzten Mal nötig.

b) Verwende statt w einen vollen Hilfsvektor nur aus Integern (Komponenteninformation) $\longrightarrow$ Übungsblatt 1

1.1.6 Algorithmus (b)

$SAXPY(n, ind_x, ind_y, val_x, val_y, nnz(x), nnz(y), \alpha)$

{Berechnet bei gepackten Vektoren x, y ein SAXPY $y := \alpha x + y$. Das Resultat steht in dem vollen Vektor w zur Verfügung und wird erst am Ende in y geschrieben.}

{innerhalb eines GAXPYs nur beim ersten SAXPY notwendig}
$scatter\ (w, n, ind_y, val_y, nnz(y))$ {w enthält y}
$j := nnz(y)$
for $i = 1$ to $nnz(x)$ **do** {Lauf über x}
  if $w_{ind_x(i)} \neq 0$ **then**
    $w_{ind_x(i)} := \alpha \cdot val_x(i) + w_{ind_x(i)}$
  else
    $w_{ind_x(i)} = \alpha \cdot val_x(i)$
    $j++$
    $ind_y(j) = ind_x(i)$ {$struct(y)$ ist bekannt, die Werte nicht}
  end if
end for
$nnz(y) := j$

{innerhalb eines GAXPYs nur beim letzten SAXPY notwendig}
for $i = 1$ to $nnz(y)$ **do** {Lauf über y, sammelt die Werte ein}
  $val_y(i) := w_{ind_y(i)}$
  $w_{ind_y(i)} := 0$
end for

Aufwand: $\mathcal{O}(nnz(x) + nnz(y))$, bei einfachem SAXPY etwas aufwendiger als Algorithmus 1.1.6.

1.1.7 Algorithmus (a)

$ddot(n, ind_x, ind_y, va\ell_x, va\ell_y, nnz(x), nnz(y), \gamma)$

{Berechnet $\gamma = x^T y$ für Vektoren x, y in gepacktem Format; verwendet Hilfsvektor w wie in Algorithmus 1.1.6}

$\gamma := 0$
$scatter\ (w, n, ind_x, va\ell_x, nnz(x))$
for $i = 1$ to $nnz(y)$ **do** {Lauf über y}
  $\gamma := \gamma + w_{ind_y(i)} \cdot va\ell_y(i)$
end for
setze w zurück auf 0 {Lauf über x}

Sind ind_x und ind_y geordnet, so kann man auf w ganz verzichten.

1.1.7 Algorithmus (b)

$ddot(n, ind_x, ind_y, va\ell_x, va\ell_y, nnz(x), nnz(y), \gamma)$

{Berechnet $\gamma = x^T y$ für Vektoren x, y in gepacktem Format; setzt ind_x und ind_y als geordnet voraus}

$\gamma := 0$; $j := 1$
for $i = 1$ to $nnz(x)$ **do**
  while $ind_y(j) < ind_x(i)$ **do**
    $j{+}{+}$
  end while
  if $ind_y(j) = ind_x(i)$ **then**
    $\gamma := \gamma + va\ell_x(i) \cdot va\ell_y(j)$
  end if
end for

Abschnitt 1.2

Dünn besetzte Matrizen

Für eine Matrix $A \in \mathbb{R}^{n \times n}$ bezeichnet

$$\begin{aligned} struct(A) &= \{(i,j) \mid a_{ij} \neq 0\} \subseteq \{1, \dots, n\}^2 \\ nnz(A) &= |struct(A)| \end{aligned}$$

1.2.1 Definition
Das *Koordinatenformat* für eine dünn besetzte Matrix $A = (a_{ij}) \in \mathbb{R}^{n \times n}$ ist ein Tripel $(rn, cn, va\ell)$ mit $rn, cn \in \mathbb{N}^{nnz(A)}$ und $va\ell \in \mathbb{R}^{nnz(A)}$, so dass die Abbildung

$$k \in \{1, \dots, nnz(A)\} \mapsto (rn(k), cn(k)) \in struct(A)$$

bijektiv ist und

$$(0 \ \neq\ )\ va\ell(k) = a_{rn(k),cn(k)},\ k = 1, \dots, nnz(A).$$

1.2.2 Beispiel

$$A = \begin{pmatrix} 1.0 & 0 & 0 & -1.0 & 0 \\ 2.0 & 0 & -2.0 & 0 & 3.0 \\ 0 & -3.0 & 0 & 0 & 0 \\ 0 & 4.0 & 0 & -4.0 & 0 \\ 5.0 & 0 & -5.0 & 0 & 6.0 \end{pmatrix}$$

k	1	2	3	4	5	6	7	8	9	10	11
rn	1	1	2	2	2	3	4	4	5	5	5
cn	1	4	1	3	5	2	2	4	1	3	5
$va\ell$	1.0	-1.0	2.0	-2.0	3.0	-3.0	4.0	-4.0	5.0	-5.0	6.0

rn : „ row numbers “

cn : „ column numbers “

Beobachtungen: Der Speicherbedarf beträgt $3nnz(A) + \mathcal{O}(1)$. cn und rn können ungeordnet werden. Dann sind Zeilen/Spalten nur sehr umständlich zu finden. Wie findet man z.B. Spalte 2 in Beispiel 1.2.2?

Vereinbarungen:

a) Wir nutzen MATLAB-Notation für Teile von Vektoren und Matrizen, z.B.

$$x(4:8) = (x_4, x_5, x_6, x_7, x_8)$$

und $A_{i\cdot} = A(i,:)$ für die i-te Zeile von A sowie $A_{\cdot j} = A(:,j)$ für die j-te Spalte von A.

b) Wir gehen im Allgemeinen davon aus, dass A keine Nullzeile oder Nullspalte besitzt (sonst wäre A singulär). Wir ersparen uns so Spezialfälle beim Handling „leerer“ Datenstrukturen.

1.2.3 Definition

Für $A = (a_{ij}) \in \mathbb{R}^{n \times n}$ ist das *zeilenweise gepackte Format* ein Quadrupel $(r\ell, rst, cn, va\ell)$ mit

$$\begin{aligned} &r\ell, rst \in \mathbb{N}^n && \text{(row length, row start)} \\ &cn \in \mathbb{N}^{nnz(A)} \\ &va\ell \in \mathbb{R}^{nnz(A)} \end{aligned}$$

so dass gilt: für $i = 1$ bis n ist

$$\begin{aligned} &cn(rst(i) : rst(i) + r\ell(i) - 1) \\ &va\ell(rst(i) : rst(i) + r\ell(i) - 1) \end{aligned}$$

das gepackte Format für die Zeile i von A.

1.2.4 Beispiel

Zeilenweise gepacktes Format für A aus Beispiel 1.2.2

k/i	1	2	3	4	5	6	7	8	9	10	11
rst	1	3	6	7	9						
$r\ell$	2	3	1	2	3						
cn	1	4	1	3	5	2	2	4	1	3	5
$va\ell$	1.0	-1.0	2.0	-2.0	3.0	-3.0	4.0	-4.0	5.0	-5.0	6.0

Beachte:

a) Der Speicherdarf beträgt nur noch $2nnz(A) + \mathcal{O}(n)$.

b) Weder Zeilen noch Spalten brauchen geordnet zu sein.

c) Sind die Zeilen geordnet, so kann man auf $r\ell$ verzichten (denn $r\ell(i) = rst(i+1) - rst(i)$), wenn man $rst(n+1) = rst(n) + r\ell(n)$ zusätzlich verwendet. $r\ell$ ist jedoch wichtig als Information, wenn die Zeilen ungeordnet sind.

d) Betrachte SAXPY-Operation $A_{i\cdot} \leftarrow \alpha A_{j\cdot} + A_{i\cdot}$ auf den Zeilen von A. (Die benötigt man bei der Gauß-Elimination.) Dann vergrößert sich im Allgemeinen $struct(A_{i\cdot})$. Dann speichert man das neue $A_{i\cdot}$ am Ende von cn und $va\ell$ und ändert $r\ell(i), rst(i)$ entsprechend. $\rightsquigarrow$ Zeilen werden ungeordnet, es entstehen „Löcher" in cn und $va\ell$ ($\widehat{=}$ altes $A_{i\cdot}$)

Abhilfe: „Komprimieren" nach einer bestimmten Zahl solcher SAXPY's.

Oder: Verwalte jede Zeile als Liste.

Vereinbarung: (wegen MATLAB, FORTRAN):

Verkettete Listen werden immer durch Felder realisiert (keine expliziten Zeiger!)

Beispiel: Einfach verkettete Liste.

i	1	2	3	4
$va\ell$	10.0	3.0	5.0	2.1
$next$	2	3	4	0
$header$	1			

$header$**:** Listenanfang

$next$**:** Verweis auf Index i des nächsten Listenelements

$va\ell$**:** Werte der Listenelemente

Andere Möglichkeit für die gleiche Liste

i	1	2	3	4
$va\ell$	3.0	10.0	2.1	5.0
$next$	4	1	0	3
$header$	2			

Einfügen oder löschen eines Listenelements benötigt einen Aufwand

$$\mathcal{O}(\text{Länge}) \text{ (Auffinden der Position)} + \mathcal{O}(1) \text{ (für Einfügen/Löschen).}$$

Beispiel: Füge den Wert 4.0 hinter 3.0 ein in obiger Liste.

i	1	2	3	4	5
$va\ell$	3.0	10.0	2.1	5.0	4.0
$next$	5	1	0	3	4
$header$	2				

Beachte: Spielt die Anordnung keine Rolle, so kann immer am Anfang mit Aufwand $\mathcal{O}(1)$ eingefügt werden. (*header* ändert sich dabei.)

1.2.5 Definition
Für $A = (a_{ij}) \in \mathbb{R}^{n \times n}$ ist das *(zeilenweise) Listenformat* gegeben durch ein Quadrupel $(rst, cn, va\ell, \ell inks)$ mit

$$cn, \ell inks \in \mathbb{N}^{nnz(A)}$$

$$rst \in \mathbb{N}^n$$

$$va\ell \in \mathbb{R}^{nnz(A)}$$

so dass gilt:

> für $i = 1$ bis n ist $rst(i)$ Header einer verketteten Liste für die Nichtnulleinträge von $A_{i\cdot}$ mit Spaltennummern in cn und Werten in $va\ell$.

Präzise bestimmt man

$k = 1, \quad j_k = rst(i)$

while $j_k \neq 0$ **do**
  $j_{k+1} = \ell inks(j_k)$
  $k++$
end while
$r\ell(i) = k - 1$

so gilt $a_{i,cn(j_k)} = va\ell(j_k), \; k = 1, \ldots, r\ell(i)$.

1.2.6 Beispiel
Matrix aus Beispiel 1.2.2 in Listenformat

k/i	1	2	3	4	5	6	7	8	9	10	11
rst	1	3	6	7	9						
cn	1	4	1	3	5	2	2	4	1	3	5
$va\ell$	1.0	-1.0	2.0	-2.0	3.0	-3.0	4.0	-4.0	5.0	-5.0	6.0
$\ell inks$	2	0	4	5	0	0	8	0	10	11	0

Ein weiteres Beispiel für ein Listenformat für A aus Beispiel 1.2.2:

k/i	1	2	3	4	5	6	7	8	9	10	11
rst	3	1	8	10	5						
cn	5	3	4	5	3	1	1	2	4	2	1
$va\ell$	3.0	-2.0	-1.0	6.0	-5.0	5.0	2.0	-3.0	-4.0	4.0	1.0
$\ell inks$	2	7	11	6	4	0	0	0	0	9	0

Bekannt aus Informatik II: Eine „ordentliche“ Listenverwaltung „merkt“ sich gelöschte Positionen in einer extra Liste. So ist immer bekannt, wo noch Speicher zur Verfügung steht.

Transformation zwischen den verschiedenen Formaten
typisch: Konversion Koordinatenformat → gepacktes Format oder Listenformat
atypisch: Konversion auf Koordinatenformat. (aber: Visualisierung mit `spy` in Matlab)

Konversion (ungeordnetes) Koordinatenformat $(rn, cn, va\ell)$ **in Listenformat** $(rst, cn, va\ell, \ell inks)$
Bestimme $\ell inks$ und rst aus rn. Die Felder cn und $va\ell$ bleiben unverändert.
Vorgehen: Für jedes i: Baue aus einer leeren Liste für Zeile i die gesamte Zeile auf durch Einfügen am Anfang.

1.2.7 Algorithmus
{Konvertiert Koordinatenformat in Listenformat}

for $i = 1$ to n **do**
  $rst(i) := 0$
end for
for $k = 1$ to $nnz(A)$ **do**
  $i := rn(k)$ { neues Element für Zeile i }
  $\ell inks(k) := rst(i)$ { Einfügen am Anfang }
  $rst(i) := k$
end for

Aufwand: $\mathcal{O}(nnz(A)) + \mathcal{O}(n)$

Konversion Listenformat $(rst\ell, cn\ell, val\ell, \ell inks)$ **in gepacktes Format** $(rst, r\ell, cn, val)$

1.2.8 Algorithmus
{konvertiert Listenformat in gepacktes Format }

$k := 1$
for $i = 1$ to n **do**
  $j := rst\ell(i)$
  $r\ell(i) := 0$
  $rst(i) := k$
  while $j \neq 0$ **do**
    $cn(k) := cn\ell(j)$
    $val(k) := val\ell(j)$
    $r\ell(i){+}{+}$
    $k{+}{+}$
    $j := \ell inks(j)$
  end while
end for

Aufwand: $\mathcal{O}(nnz(A)) + \mathcal{O}(n)$

Die Konversion von gepacktem Format nach Koordinatenformat ist „straightforward“, man erhält aber zunächst ungeordnete Zeilen. $\rightsquigarrow$ Sortieralgorithmus verwenden.

Bei (zeilenweise) gepacktem Format und Listenformat ist der Zugriff auf eine Zeile gut möglich (und für die Gauß-Elimination auch notwendig). Allerdings: Bei der Elimination von Spalte k müssen nur die Zeilen modifiziert werden, die in Spalte k besetzt sind. Zumindest $struct(A)$ sollte auch spaltenweise zugänglich sein.

Lösung:

- gepacktes Format: Verwende zusätzlich ein spaltenweise gepacktes Format mit $cst, c\ell, rn$:

  Beispiel: Matrix aus Beispiel 1.2.2

k/i	1	2	3	4	5	6	7	8	9	10	11
rst	1	3	6	7	9						
$r\ell$	2	3	1	2	3						
cn	1	4	1	3	5	2	2	4	1	3	5
$va\ell$	1.0	-1.0	2.0	-2.0	3.0	-3.0	4.0	-4.0	5.0	-5.0	6.0
cst	1	4	6	8	10						
$c\ell$	3	2	2	2	2						
rn	1	2	5	3	4	2	5	1	4	2	5

- Listenformat: Verwende gleichzeitig spaltenweises Listenformat (ohne $va\ell$) also nur $rn, \ell inkscol, cst$. Dabei bezeichnet $\ell inkscol$ die Links für die Spaltenlisten.

  Beispiel: siehe Übungsblatt 2.

1.2.9 Bemerkung

a) Die beschriebenen erweiterten Formate (gepackt oder Liste) werden unsere Standardformate sein.

b) Die Spalteninformation kann einfach aus den Informationen des nicht erweiterten Formates bestimmt werden mit Aufwand $\mathcal{O}(nnz(A))$. $\rightsquigarrow$ Übungsaufgabe.

Zeilenvertauschungen kommen bei der Spaltenpivotsuche bei der Gauß-Elimination vor. Realisierung bei unseren Standardformaten: vertausche Zeilen i_1 und i_2:

- erweitertes gepacktes Format:

  vertausche $rst(i_1)$ mit $rst(i_2)$ und $r\ell(i_1)$ mit $r\ell(i_2)$

sowie alle i_1 und i_2 in rn. Dabei müssen in rn nur die Abschnitte durchlaufen werden, die Spalten beschreiben, welche in den Zeilen i_1 und i_2 besetzt sind. S. Algorithmus 1.2.10

- erweitertes Listenformat:

  vertausche $rst(i_1)$ mit $rst(i_2)$

  sowie alle i_1 und i_2 in rn. Dabei müssen in rn nur die Listen durchlaufen werden, die Spalten beschreiben, welche in den Zeilen i_1 und i_2 besetzt sind.

1.2.10 Algorithmus

{vertauscht Zeilen i_1 und i_2 im erweiterten gepackten Format}

```
vertausche rst(i1) und rst(i2) sowie rℓ(i1) und rℓ(i2)
for k = 0 to rℓ(i1) − 1 do                          {Lauf über Zeile i1}
  j := cn(rst(i1) + k)                    {Spalte j ist in Zeile i1 besetzt}
  for ℓ = 0 to cℓ(j) − 1 do                           {In Spalte j ...}
    i = rn(cst(j) + ℓ)
    if i = i1 then                         {...wird Zeile i1 durch i2 ...}
      rn(cst(j) + ℓ) = i2
    end if
    if i = i2 then                           {...und i2 durch i1 ersetzt}
      rn(cst(j) + ℓ) = i1
    end if
  end for
end for
for k = 0 to rℓ(i2) − 1 do                {jetzt dasselbe für Zeile i2}
  j := cn(rst(i2) + k)          {kommen i1 und i2 beide in Spalte j vor ...}
  for ℓ = 0 to cℓ(j) − 1 do  {...wurden sie vorher schon vertauscht ...}
    i = rn(cst(j) + ℓ)                              {...und jetzt nochmal}
    if i = i1 then
      rn(cst(j) + ℓ) = i2
    end if
    if i = i2 then
      rn(cst(j) + ℓ) = i1
    end if
  end for
end for
```

Aufwand: Falls $|struct(A(i,:)| = \mathcal{O}(c)$ und $|struct(A_{:,j}| = \mathcal{O}(c)$: $\mathcal{O}(c^2)$

Allgemeine Permutationen

1.2.11 Definition
Sei π eine Permutation. Die zugehörige *Permutationsmatrix* erfüllt.

$$(Px)_{\pi(i)} = x_i \ , i = 1, \ldots, n \text{ für alle } x \in \mathbb{R}^n.$$

Dann ist PA eine Zeilenpermutation von A. PA erhält man in unseren Formaten einfach so:

ersetze $rst(i)$ durch $rst(\pi(i))$, $\quad i = 1, \ldots, n$
ersetze $r\ell(i)$ durch $r\ell(\pi(i))$, $\quad i = 1, \ldots, n$
sowie in rn jedes i durch $\pi(i)$, $\quad i = 1, \ldots, n$

Aufwand: $\mathcal{O}(n) + \mathcal{O}(nnz(A))$

AP^T ist Permutation der Spalten. Man erhält AP^T aus A durch analoge Modifikation. Zeilen (cn) sind dann ungeordnet. $A \to PAP^T$ entsprechend.

Zur Übung: Entwickle Algorithmus zur Berechnung von $y = Ax$, x, A, y in gepacktem Format:

$$\begin{aligned} A &= [rst, r\ell, cn, cst, c\ell, rn, va\ell] \\ x &= [indx, va\ell x] \end{aligned}$$

1.2.12 Algorithmus
{Berechnet $y = Ax$ mit x, y, A gepackt; verwendet Hilfsvektor w voller Länge}

$m := 0$
$scatter(w, n, ind_x, va\ell_x, nnz(x))$
for $i = 1$ to n **do**
  $\gamma := 0$
  for $j = 0$ to $r\ell(i) - 1$ **do**
    $k := rst(i) + j$
    $\gamma := \gamma + va\ell(k) \cdot w(cn(k))$
  end for
  if $\gamma \neq 0$ **then**
    $m := m + 1$
    $va\ell_y(m) := \gamma$
    $ind_y(m) := i$
  end if
end for

Beachte:

a) Zufällige Nullen werden (ausnahmsweise) berücksichtigt.

b) Für den Aufwand gilt: $\mathcal{O}(nnz(A))$. Setze $c = \frac{nnz(A)}{n}$ (mittlere Zahl von Nichtnullen pro Zeile), so ist der Aufwand also $\mathcal{O}(c \cdot n)$. Zum Vergleich: A vollbesetzt $\rightsquigarrow \mathcal{O}(n^2)$.

Alternative: spaltenorientiert

$$Ax = \sum_{j=1;\ j \in struct(x)}^{n} A_{\cdot j} \cdot x_j$$

alternativer Algorithmus damit

$y := 0$
for $k = 1$ to $nnz(x)$ **do**
  $y := y + A_{\cdot, ind_x(k)} va\ell_x(k)$ $\leftarrow$ SAXPY
end for

Hierfür ist die volle Information zur Spalte $ind_x(k)$ (inklusive der Werte!) notwendig, was bei unseren Standardformaten nicht der Fall ist.

1.2.13 Definition

a) Sind ind_1 und ind_2 zwei Indexvektoren $\subseteq \{1, \ldots, n\}$ und ist $A \in \mathbb{R}^{n \times n}$, so bezeichnet $B = A(ind_1, ind_2)$ die Matrix (b_{ij}) mit

$$b_{ij} = A(ind_1(i), ind_2(j)) \quad , i = 1, \ldots, |ind_1|, j = 1, \ldots, |ind_2|$$

(wie in MATLAB)

b) Das *Cliquenformat* einer Matrix $A \in \mathbb{R}^{n \times n}$ ist eine Sammlung von Matrizen $A_k \in \mathbb{R}^{n_k \times n_k}$ und Indexvektoren s_k der Länge $n_k, k = 1, \ldots, m$, so dass gilt

$$A = \sum_{k=1}^{m} B_k \text{ mit } B_k(s_k, s_k) = A_k, struct(B_k) \subseteq s_k \times s_k.$$

Die Matrizen A_k heißen *Cliquen.*

Cliquen entstehen bei der Gauß-Elimination von symmetrischen Matrizen in natürlicher Weise.

Abschnitt 1.3

Die LDU-Zerlegung für volle Matrizen

(eine Wiederholung)

Im ganzen Abschnitt sei $A \in \mathbb{R}^{n \times n}$ regulär.

1.3.1 Definition
Die LDU-Zerlegung von A ist eine Darstellung

$$A = LDU, \quad L, D, U \in \mathbb{R}^{n \times n}$$

wobei L linke untere Dreiecksmatrix mit Einheitsdiagonale, D Diagonalmatrix, U rechte obere Dreiecksmatrix mit Einheitsdiagonale ist.

$$L = \begin{bmatrix} 1 & & & & 0 \\ \ell_{2,1} & 1 & & & \\ \vdots & \ddots & \ddots & & \\ \vdots & & \ddots & 1 & \\ \ell_{n,1} & \cdots & \cdots & \ell_{n,n-1} & 1 \end{bmatrix}, \quad D = \begin{bmatrix} d_{11} & & & 0 \\ & \ddots & & \\ & & \ddots & \\ & & & \ddots \\ 0 & & & d_{nn} \end{bmatrix},$$

$$U = \begin{bmatrix} 1 & u_{1,2} & \cdots & \cdots & u_{1,n} \\ & 1 & \ddots & & \vdots \\ & & \ddots & \ddots & \vdots \\ & & & 1 & u_{n-1,n} \\ 0 & & & & 1 \end{bmatrix}$$

1.3.2 Bemerkung

a) Die LDU-Zerlegung existiert nicht immer, z.B.

$$A = \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ \ell_{21} & 1 \end{pmatrix} \begin{pmatrix} d_{11} & 0 \\ 0 & d_{22} \end{pmatrix} \begin{pmatrix} 1 & u_{12} \\ 0 & 1 \end{pmatrix}$$

$$\Rightarrow d_{11} = 0 \qquad \text{(aus Pos } (1,1)\text{)}$$

$$\Rightarrow 1 = d_{11} u_{12} \text{ (aus Pos } (1,2)\text{) nicht erfüllbar.}$$

b) Im Falle der Existenz ist die LDU-Zerlegung eindeutig. ($\rightsquigarrow$ siehe Satz 1.3.5)

1.3.3 Algorithmus (Gauß-Elimination $(A = A^{(1)})$)

for $k = 1$ to $n-1$ **do** $\{$Spaltenpivotsuche$\}$
  bestimme s_k mit $|a_{s_k k}^{(k)}| = \max_{i=k}^{n} |a_{ik}^{(k)}|$
  vertausche Zeilen k und s_k in $A^{(k)}$, ergibt $\tilde{A}^{(k)}$.
  for $i = k+1$ to n **do** $\{$Zeilenfaktoren$\}$
    $\ell_{ik} = \tilde{a}_{ij}^{(k)} / \tilde{a}_{kk}^{(k)}$
  end for
  for $i = k+1$ to n **do** $\{$restliche Zeilen$\}$
    for $j = k+1$ to n **do** $\{$restliche Spalten$\}$
      $a_{ij}^{(k+1)} = \tilde{a}_{ij}^{(k)} - \ell_{ik}\tilde{a}_{kj}^{(k)}$
    end for
  end for
  (setze $d_{kk} = \tilde{a}_{kk}^{(k)}, u_{kj} = \tilde{a}_{kj}^{(k)}/d_{kk}, j = k+1, \ldots, n$)
end for

1.3.4 Lemma
Die Gauß-Elimination erfordert einen Aufwand von

Multiplikationen:	$\frac{1}{3}n^3 + \mathcal{O}(n^2)$
Additionen:	$\frac{1}{3}n^3 + \mathcal{O}(n^2)$
Divisionen:	$\frac{n^2}{2} + \mathcal{O}(n)$
	($n^2 + \mathcal{O}(n)$ bei Bestimmung von D und U)

Beweis:
Multiplikationen:

$$\begin{aligned}\sum_{k=1}^{n-1}\sum_{i=k+1}^{n}\sum_{j=k+1}^{n} 1 &= \sum_{k=1}^{n-1}(n-k)^2 \\ &= \sum_{k=1}^{n-1} k^2 \\ &= \frac{(n-1)n(2n-1)}{6} = \frac{1}{3}n^3 + \mathcal{O}(n^2).\end{aligned}$$

Additionen: Genauso.
Divisionen:

$$\sum_{k=1}^{n-1}\sum_{i=k+1}^{n} 1 = \sum_{k=1}^{n-1}(n-k) = \sum_{k=1}^{n-1} k = \frac{1}{2}n^2 + \mathcal{O}(n).$$

□

1.3.5 Satz
Es sei

$$L^{(k)} = \begin{bmatrix} 1 & & & & & \\ & 1 & & & & \\ & & 1 & & & \\ & & \ell_{k+1,k} & 1 & & \\ & & \vdots & & \ddots & \\ & & \ell_{n,k} & 0 & & 1 \end{bmatrix}, D^{(k)} = \begin{bmatrix} d_{11} & & & & & \\ & \ddots & & & & \\ & & d_{kk} & & & \\ & & & 1 & & \\ & & & & \ddots & \\ & & & & & 1 \end{bmatrix},$$

$$U^{(k)} = \begin{bmatrix} 1 & u_{1,2} & & \dots & & u_{1,n} \\ & \ddots & \ddots & & & \vdots \\ & & 1 & u_{k,k+1} & \dots & u_{k,n} \\ & & & \tilde{a}^{(k)}_{k+1,k+1} & \dots & \tilde{a}^{(k)}_{k+1,n} \\ & & & \vdots & & \vdots \\ & & & \tilde{a}^{(k)}_{n,k+1} & \dots & \tilde{a}^{(k)}_{n,n} \end{bmatrix}, \quad A^{(k)} = D^{(k)}U^{(k)}$$

$P^{(k)} \in \mathbb{R}^{n\times n}$ Permutationsmatrix, welche k und s_k vertauscht

$$(\Rightarrow (P^{(k)})^{-1} = P^{(k)}).$$

Dann gilt:

$$P^{(k)}A^{(k)} = L^{(k)}A^{(k+1)}, k = 1, \dots, n-1 \tag{1.3.1}$$

und

$$PA = LDU \tag{1.3.2}$$

mit

$$\begin{aligned} L &= \hat{L}^{(1)} \dots \hat{L}^{(n-1)}, \\ \hat{L}^{(k)} &= \left[Q^{(k)}\right]^{-1} L^{(k)}Q^{(k)}, \\ Q^{(k)} &= P^{(k+1)} \dots P^{(n-1)},\ k = 1, \dots, n-1 \qquad (Q^{(n-1)} = I), \\ U &= U^{(n)},\ D^{(k)}U^{(k)} = A^{(k)},\ D = D^{(n)}, \\ P &= P^{(n-1)} \dots P^{(1)}. \end{aligned}$$

Beweis: Es gilt

$$(L^{(k)})^{-1} = \begin{bmatrix} 1 & & & & & \\ & \ddots & & & & \\ & & 1 & & & \\ & & -\ell_{k+1,k} & \ddots & & \\ & & \vdots & & \ddots & \\ & & -\ell_{n,k} & & & 1 \end{bmatrix} \text{ und } P^{(k)}A^{(k)} = \tilde{A}^{(k)}.$$

Damit ergibt sich aus Algorithmus 1.3.3:

$$\begin{aligned} A^{(k+1)} &= (L^{(k)})^{-1}\tilde{A}^{(k)} = (L^{(k)})^{-1}P^{(k)}A^{(k)} \\ \Longleftrightarrow L^{(k)}A^{(k+1)} &= P^{(k)}A^{(k)} \Longleftrightarrow (1.3.1). \end{aligned}$$

Zum Beweis von (1.3.2): Es gilt zunächst (nach (1.3.1))

$$A^{(n-1)} = P^{(n-1)}L^{(n-1)}A^{(n)} = P^{(n-1)}\hat{L}^{(n-1)}DU$$

$$\begin{aligned} \Rightarrow A^{(n-2)} &= P^{(n-2)}L^{(n-2)}A^{(n-1)} \\ &= P^{(n-2)}L^{(n-2)}P^{(n-1)}\hat{L}^{(n-1)}DU \\ &= P^{(n-2)}Q^{(n-2)}[Q^{(n-2)}]^{-1}L^{(n-2)}Q^{(n-2)}\hat{L}^{(n-1)}DU \\ &= P^{(n-2)}Q^{(n-2)}\hat{L}^{(n-2)}\hat{L}^{(n-1)}DU \\ &= P^{(n-2)}P^{(n-1)}\hat{L}^{(n-2)}\hat{L}^{(n-1)}DU \end{aligned}$$

$$\begin{aligned} &\Rightarrow A^{(n-3)} = \ldots = P^{(n-3)}P^{(n-2)}P^{(n-1)}\hat{L}^{(n-3)}\hat{L}^{(n-2)}\hat{L}^{(n-1)}DU \\ &\Rightarrow \quad A = A^{(1)} = P^{(1)}\ldots P^{(n-1)}\hat{L}^{(1)}\ldots\hat{L}^{(n-1)}DU \\ &\Rightarrow \quad P^{(n-1)}\ldots P^{(1)}A = \hat{L}^{(1)}\ldots\hat{L}^{(n-1)}DU \\ &\Rightarrow \quad PA = LDU. \end{aligned}$$

□

1.3.6 Bemerkung

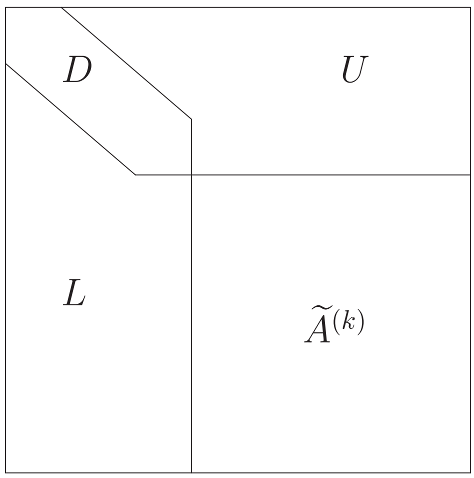
a) $Q^{(k)}$ entspricht einer Permutation nur auf $\{k+1, k+2, \ldots, n-1\}$. Es ist deshalb

$$\hat{L}^{(k)} = [Q^{(k)}]^{-1} L^{(k)} Q^{(k)} = \begin{bmatrix} 1 & & & & & \\ & \ddots & & & & \\ & & 1 & & & \\ & & \hat{\ell}_{k+1,k} & \ddots & & \\ & & \vdots & & \ddots & \\ & & \hat{\ell}_{n,k} & & & 1 \end{bmatrix}$$

mit $\hat{L}^{(k)}_{\cdot k} = [Q^{(k)}]^{-1} L^{(k)}_{\cdot k}$. Also ist insbesondere

$$L = \hat{L}^{(1)} \dots \hat{L}^{(m-1)} = \begin{bmatrix} 1 & & & \\ \hat{\ell}_{2,1} & \ddots & & \\ \vdots & \ddots & \ddots & \\ \hat{\ell}_{n,1} & \dots & \hat{\ell}_{n,n-1} & 1 \end{bmatrix}.$$

b) Bei Implementierungen benötigt man nur ein (zweidimensionales) Feld A als Speicher; in Schritt k ist es dann wie folgt belegt:



Zusätzlich muss man sich die angewandten Permutationen in einem Indexvektor merken.

1.3.7 Satz
A regulär $\Longleftrightarrow$ $\tilde{a}_{kk} \neq 0$ für $k = 1, \dots, n$. Insbesondere ist Gauß-Elimination durchführbar, wenn A regulär ist.

Beweis:

„$\Leftarrow$“ Algorithmus 1.3.3 durchführbar, d.h. $\tilde{a}_{kk} \neq 0$ für $k = 1, \ldots, n$. Es folgt $PA = LDU \iff A = P^{-1}LDU$ ist regulär. (Da P^{-1}, L, D, U regulär).

„$\Rightarrow$“ Angenommen, es existiert (kleinstes) k mit $\tilde{a}_{kk}^{(k)} = 0$ Dann ist $\tilde{a}_{ik}^{(k)} = 0$ für $i = k, \ldots, n$ und damit die Matrix

$$\tilde{A}^{(k)} = \begin{bmatrix} * & * & * & \square & \square & \square \\ & * & * & \square & \square & \square \\ 0 & & * & \square & \square & \square \\ 0 & \ldots & \ldots & 0 & \square & \square \\ & & & \vdots & \square & \square \\ & & & 0 & \square & \square \end{bmatrix}$$

singulär, denn Spalte k ist linear abhängig von Spalten 1 bis $k-1$. Damit ist auch $A^{(k)}$ singulär und wegen (1) auch $A^{(k-1)}, A^{(k-2)}, \ldots, A^{(1)}$.

$\square$

1.3.8 Bemerkung

a) „Verschiedene“ Dreieckszerlegungen entstehen durch verschiedene „Klammerung“ in $PA = LDU$, z.B.

$$PA = L(DU) \qquad \text{„Gauß-Zerlegung“}$$

$$A = A^T : \begin{cases} A = (LD^{\frac{1}{2}})(D^{\frac{1}{2}}U) & \text{„Cholesky-Zerlegung“}(U = L^T) \\ A = LDU, U = L^T & \text{„wurzelfreie Cholesky-Zerlegung“.} \end{cases}$$

b) Die Berechnung der LDU-Zerlegung $PA = LDU$ ist der aufwändige Bestandteil eines direkten Gleichungslösers. Man erhält x in $Ax = b$ durch

$$\begin{array}{lll} & & PAx = Pb \\ \iff & & LDUx = Pb \\ \iff & Lz = Pb & \text{„Dreieckssystem“, } \mathcal{O}(n^2) \\ & Dy = z & \text{„Diagonalsystem“, } \mathcal{O}(n) \\ & Ux = y & \text{„Dreieckssystem“, } \mathcal{O}(n^2). \end{array}$$

c) Zur Vereinfachung formulieren wir jetzt weitere Faktorisierungsalgorithmen **ohne** Pivotisierung, d.h. es ist $PA = A = LDU$.

Varianten der Gauß-Elimination entstehen, wenn man die Einträge in L, D, U in anderer Reihenfolge berechnet als bei Gauß-Elimination. Dabei ändert sich der Zugriff auf die Daten von $A \rightsquigarrow$ unterschiedliche Eignung für unsere Datenformate. Wir gehen systematisch vor:

$$A = LDU$$

ergibt

$$a_{ij} = \sum_{k=1}^{\min\{i,j\}} \ell_{ik} d_{kk} u_{kj}, \quad i, j = 1, \ldots, n$$

also durch Auflösen

$$d_{ii} = a_{ii} - \sum_{k=1}^{i-1} \ell_{ik} d_{kk} u_{ki} \quad (i = j) \tag{1.3.3}$$

$$\ell_{ij} = \left(a_{ij} - \sum_{k=1}^{j-1} \ell_{ik} d_{kk} u_{kj} \right) / d_{jj} \quad (i > j) \tag{1.3.4}$$

$$u_{ij} = \left(a_{ij} - \sum_{k=1}^{i-1} \ell_{ik} d_{kk} u_{kj} \right) / d_{ii} \quad (i < j). \tag{1.3.5}$$

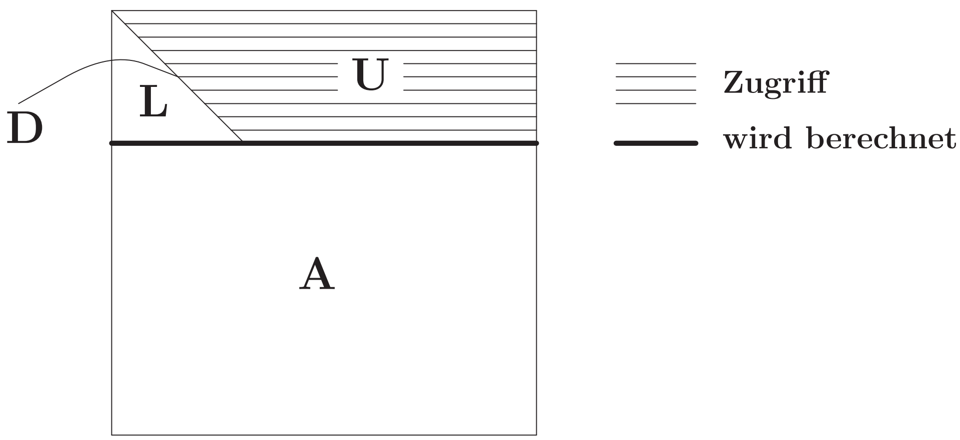
Man kann (1.3.3) - (1.3.5) also direkt zur Bestimmung von L, D, U verwenden, vorausgesetzt, in die rechten Seiten gehen nur bekannte Größen ein.

1. Möglichkeit: Zeilenweise durch D, L und U.

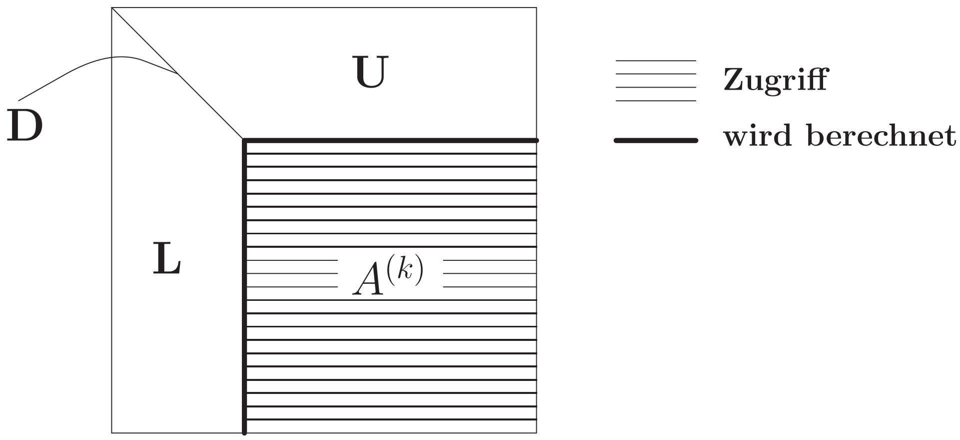
1.3.9 Algorithmus (Doolittle-Algorithmus)

for $i = 1$ to n **do**
  for $j = 1$ to $i - 1$ **do**
    $\ell_{ij} = \left(a_{ij} - \sum_{k=1}^{j-1} \ell_{ik} d_{kk} u_{kj} \right) / d_{jj}$ $\{i$-te Zeile von $L\}$
  end for
  $d_{ii} = a_{ii} - \sum_{k=1}^{i-1} \ell_{ik} d_{kk} u_{ki}$ $\{i$-te Zeile von $D\}$
  for $j = i + 1$ to n **do**
    $u_{ij} = \left(a_{ij} - \sum_{k=1}^{i-1} \ell_{ik} d_{kk} u_{kj} \right) / d_{ii}$ $\{i$-te Zeile von $U\}$
  end for
end for

Skizze für Schritt i:



Zum Vergleich: Schema bei Gauß-Elimination („kij-Form“).
Skizze für Schritt k:

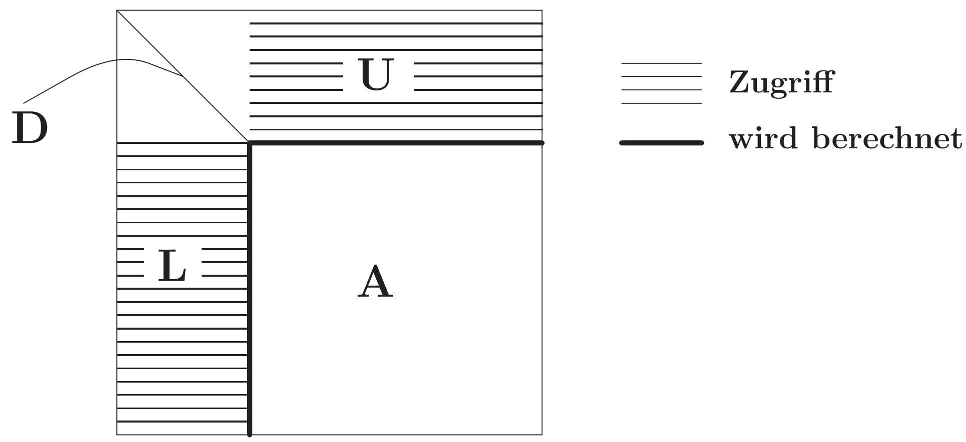


2. Möglichkeit: Spalten von L, Zeilen von U.

1.3.10 Algorithmus (Crout-Algorithmus)

for $i = 1$ to n **do**
  $d_{ii} = a_{ii} - \sum\limits_{k=1}^{i-1} \ell_{ik} d_{kk} u_{ki}$
  for $s = i+1$ to n **do**
    $\ell_{si} = \left(a_{si} - \sum\limits_{k=1}^{i-1} \ell_{sk} d_{kk} u_{ki} \right) / d_{ii}$
  end for
  for $j = i+1$ to n **do**
    $u_{ij} = \left(a_{ij} - \sum\limits_{k=1}^{i-1} \ell_{ik} d_{kk} u_{kj} \right) / d_{ii}$
  end for
end for

Skizze für Schritt i:



1.3.11 Algorithmus
(„ikj-Form")

$d_{11} = a_{11}$
for $i = 2$ to n **do**
  for $s = i$ to n **do**
    $u_{i-1,s} = a_{i-1,s}^{(i-1)}/d_{i-1,i-1}$
  end for
  for $k = 1$ to $i-1$ **do**
    $\ell_{ik} = a_{ik}^{(k)}/d_{kk}$
    for $j = k+1$ to n **do**
      $a_{ij}^{(k+1)} = a_{ij}^{(k)} - \ell_{ik} d_{kk} a_{kj}^{(k)}$
    end for
  end for
  $d_{ii} = a_{ii}^{(i)}$
end for

Dieser Algorithmus berechnet alle Veränderungen von Zeile i auf ein Mal. Er ensteht aus Doolittle bzw. Gauß-Elimination durch Vertauschen der Schleifen über j und k bzw. k und i.

Schema: Wie Doolittle, aber die i-te Zeile wird mehrfach durchlaufen.

1.3.12 Bemerkung
Zu allen Algorithmen existieren entsprechende spaltenorientierte Varianten. (außer für Crout)

Diskussion der Algorithmen mit Blick auf dünn besetzte Matrizen:

Gauß-Elimination: innere Schleife (über j): SAXPY mit Zeilen von $A^{(k)}$.

- $+$: im Wesentlichen zeilenweiser Zugriff (auf $A^{(k)}$)
- $-$: die Besetztheit von $A^{(k)}$ ändert sich mit k

Doolittle: innere Schleife (über k): Innenprodukt mit Zeile von L, Spalte von U.

- $+$: zeilenweise Berechnung von L und U
- $-$: spaltenweiser Zugriff auf U
- $+$: in L, U wird stets die Besetztheit der endgültigen Faktorisierung benötigt

Crout: innere Schleife (über k): Innenprodukt mit Zeile von L, Spalte von U.

+: zeilenweise Berechnung von U, Zugriff auf L

−: spaltenweise Berechnung von L, Zugriff auf U

+: nur Besetztheit der endgültigen Faktorisierung nötig

***ikj*-Form:** innere Schleife (über j): SAXPY mit Zeilen von $A^{(k)}$.

+: zeilenweiser Zugriff auf L, U und $A^{(k)}$

−: Besetztheit von $A_{i\cdot}^{(k)}$ in Schleife über k ändert sich ständig

1.3.13 Satz
Im Falle der Existenz ist die LDU-Zerlegung eindeutig.

Beweis: Folgt direkt aus (1.3.2) - (1.3.4) bzw. Doolittle-Algorithmus. □

1.3.14 Korollar
$A \in \mathbb{R}^{n \times n}$ sei symmetrisch und es existiere die LDU-Faktorisierung $A = LDU$. Dann gilt $U = L^T$.

Beweis: $LDU = A = A^T = (LDU)^T = U^T D L^T$. Wegen Satz 1.3.13 folgt $L = U^T$ und $L^T = U$. □

Folge: Im Fall $A = A^T$ muss U gar nicht berechnet werden, die Algorithmen können entsprechend verkürzt werden. (Zugriff auf u_{kj} ersetzen durch ℓ_{jk}.)

Aus dem Doolittle-Algorithmus erhalten wir so:

1.3.15 Algorithmus (Wurzelfreie Cholesky-Zerlegung, Doolittle)

for $i = 1$ to n **do**

  for $j = 1$ to $i-1$ **do**

$$\ell_{ij} = \left(a_{ij} - \sum_{k=1}^{j-1} \ell_{ik} d_{kk} \ell_{jk} \right) / d_{jj}$$

  end for

$$d_{ii} = a_{ii} - \sum_{k=1}^{i-1} \ell_{ik}^2 d_{kk}$$

end for

Beachte: Auf A und L wird nur zeilenweise zugegriffen.

1.3.16 Satz
Ist $A = A^T$ positiv definit (spd), so existiert die LDU-Zerlegung und es ist $U = L^T$.

Beweis: Wegen Existenz: siehe Satz 1.3.21. Wegen $U = L^T$: siehe Korollar 1.3.14. □

1.3.17 Bemerkung
Ist $A = A^T$, so wird bei Spaltenpivotsuche ($A \to PA$) die Symmetrie im Allgemeinen zerstört. Symmetrie bleibt erhalten bei der Transformation $A \to PAP^T$. Damit wird nur die Reihenfolge der Diagonalelemente geändert. Falls A nicht spd ist, kann so die Existenz der LDL^T nicht gesichert werden.

1.3.18 Bemerkung
Algorithmus 1.3.15 ist **die** Vorlage für einen entsprechenden Algorithmus zur Berechnung der Cholesky-Zerlegung bei dünn besetzten Matrizen.

1.3.19 Satz (Wilkinson, 1965)
Es seien $L, \tilde{U}$ die mit Gauß-Elimination ohne Pivotsuche bestimmte L(DU)-Zerlegung von A ($\tilde{U} = DU$) auf einem Computer mit Maschinengenauigkeit ϵ. Dann gilt: Es existiert eine Matrix $H \in \mathbb{R}^{n\times n}$, $H = (h_{ij})$, so dass

$$A + H = L\tilde{U}$$

und

$$|h_{ij}| \leq 5.01\ \epsilon\ \max_{k=1}^{n} |a_{ij}^{(k)}|,\ i,j = 1,\dots,n.$$

Beweis: Siehe Numerik I oder
N. Higham: Accuracy and Stability of Numerical Algorithms, 2nd edition, SIAM, Philadelphia, 2002. □
(Dieser Satz gilt auch für die anderen Algorithmen. Es werden genau die selben Berechnungen in anderer Reihenfolge durchgeführt.)

Erinnerung an Definition von ϵ: Bezeichnet fl das Ergebnis einer Rechenoperation auf dem Computer (mit Gleitkomma-Rundung), so gilt

$$fl(a \circ b) = (a \circ b)(1 + \epsilon')$$

mit $|\epsilon'| \leq \epsilon$ für $\circ \in \{+, -, *, /\}$.
Z.B. IEEE 754-Standard: Bei 64-Bit-Darstellung ist $\epsilon = 2^{-53} \approx 10^{-16}$.

Folgerung aus Satz 5: Die Zahlen $\rho_{ij} = \max_{k=1}^{n} |a_{ij}^{(k)}|$ sind ein Maß für die (Rückwärts-) Stabilität der Gauß-Elimination. Pivotstrategien sollten so sein, dass die ρ_{ij} möglichst klein bleiben.

1.3.20 Definition
Die Zahl $\rho = \max\limits_{i,j} |\rho_{ij}|$ heißt *Wachstumsfaktor* für die Gauß-Elimination.

Spaltenpivotsuche schließt nicht aus, dass $\rho = 2^{n-1} \max\limits_{i,j} |a_{ij}|$. (siehe Übungsblatt 4)

Für spd-Matrizen gilt aber:

1.3.21 Satz
A sei spd. Dann ist $\rho = \max\limits_{i,j} |a_{ij}|$. (In diesem Sinne ist Gauß-Elimination bzw Cholesky-Faktorisierung von spd-Matizen ohne Pivotsuche stabil.)

Beweis: Wir zeigen für $B = A^{(2)}(2:n, 2:n)$

a) B ist spd

b) $\max |b_{ij}| \leq \max |a_{ij}|$

Der Satz folgt dann per Induktion.

a): Bezeichne

$$\ell = \left(1, \frac{a_{12}}{a_{11}}, \ldots, \frac{a_{1n}}{a_{11}}\right)^T = \frac{1}{a_{11}} A_{\cdot 1} = \frac{1}{a_{11}} A_{1\cdot}^T.$$

Es gilt:

$$B = (A - a_{11}\ell\ell^T)(2:n, 2:n).$$

Also ist B symmetrisch.

Zu zeigen bleibt noch $y^T B y > 0$ für alle $y \in \mathbb{R}^{n-1}, y \neq 0$. Setze dazu

$$x = \begin{pmatrix} 0 \\ y \end{pmatrix}, x \neq 0.$$

Es gilt:

$$\begin{aligned} y^T B y &= x^T (A - a_{11}\ell\ell^T) x \\ &= x^T A x - a_{11}(x^T \ell)^2 \end{aligned}$$

Andererseits gilt für $z = x - (x^T\ell, 0, \ldots, 0)^T = x - (x^T\ell)e_1 \neq 0$

$$\begin{aligned} 0 < z^T A z &= x^T A x - (x^T\ell)e_1^T A x - x^T A (x^T\ell) e_1 + (x^T\ell)^2 e_1^T A e_1 \\ &= x^T A x - (x^T\ell) a_{11} \ell^T x - (x^T\ell) a_{11} x^T \ell + (x^T\ell)^2 a_{11} \\ &= x^T A x - a_{11}(x^T\ell)^2 = y^T B y. \end{aligned}$$

b): Für $i = 2, \ldots, n$ gilt

$$0 <^{a)} b_{i-1,i-1} = a_{ii}^{(2)} = a_{ii} - a_{11}\ell_i^2 \leq a_{ii}$$

Sei $i \neq j, a_{ij}^{(2)} \neq 0$ und $y = (0, \ldots, 0, 1, 0, \ldots, 0, -sgn(a_{ij}^{(2)}), 0, \ldots, 0) \in \mathbb{R}^{n-1}$. Dann gilt wegen a):

$$\begin{aligned} 0 < y^T By &= b_{i-1,i-1} + b_{j-1,j-1} - 2b_{i-1,j-1}sgn(a_{ij}^{(2)}) \\ &= b_{i-1,i-1} + b_{j-1,j-1} - 2|b_{i-1,j-1}| \end{aligned}$$

$\Rightarrow |b_{i-1,j-1}| \leq \frac{1}{2}(b_{i-1,i-1} + b_{j-1,j-1}) \leq \rho.$

□

Konsequenz: Für A spd ist ρ unabhängig von jeder Transformation $A \to PAP^T, P$ Permutation. Diese Transformation kann bei d.b. Matrizen also ausschließlich mit Blick auf den entstehenden Rechenaufwand gewählt werden.

Blick nach vorn:

$$A = \begin{pmatrix} x & x & \ldots & x \\ x & x & 0 & \\ \vdots & 0 & \ddots & \\ x & & & x \end{pmatrix} \text{ spd } \rightsquigarrow L = \begin{pmatrix} x & & & \\ x & x & & \\ \vdots & & \ddots & \\ x & \ldots & \ldots & x \end{pmatrix} \text{ „voll“}$$

$$PAP^T = \begin{pmatrix} x & & & x \\ & \ddots & 0 & \vdots \\ & 0 & \ddots & \vdots \\ x & \ldots & \ldots & x \end{pmatrix} \text{ spd } \rightsquigarrow L = \begin{pmatrix} x & & & 0 \\ & \ddots & & \\ & 0 & \ddots & \\ x & \ldots & \ldots & x \end{pmatrix} \text{ „optimal“.}$$

KAPITEL 2

Dünn besetzte spd-Matrizen

Abschnitt 2.1

Gerüst für die Faktorisierung

Im ganzen Kapitel sei $A = A^T \in \mathbb{R}^{n \times n}$ positiv definit. Zu berechnen ist die wurzelfreie Cholesky-Zerlegung $A = LDL^T$. Dies geschieht in zwei Phasen.

1. Phase (symbolische Phase)

a) Bestimmung einer geeigneter Permutation, so dass die Faktorisierung von PAP^T geringen Aufwand verursacht

b) Erzeugung der Datenstruktur („L/D-Datenstruktur“) zur Aufnahme von L und D.

2. Phase (numerische Phase)

a) ändere die L/D-Datenstruktur so ab, dass (mindestens) die Zeileninformation geordnet ist.

b) übertrage PAP^T in L/D-Datenstruktur

c) berechne die Faktorisierung $PAP^T = LDL^T$

Konvention: Die L/D-Datenstruktur ist bei uns das zeilenweise gepackte Format (bzw. Listenformat), bestehend aus rst, $r\ell$, cn, $va\ell$ (bzw. rst, cn, $\ell inks$, $va\ell$) für das **strikte** untere Dreieck von L **plus** ein Vektor d für die Diagonale von D. ($r\ell(i) = 0$ ist jetzt möglich!)

Wir behandeln zuerst Phase 2, da wir hierfür bereits alle Elemente beisammen haben.

Abschnitt 2.2

Numerische Phase

Wir ändern zuerst die L/D-Datenstruktur, so dass Information in den Zeilen geordnet ist. Vergleiche hierzu Übungsaufgabe 6. Wir behandeln nur das gepackte Format.

2.2.1 Algorithmus
(doppelte symbolische Transposition)

{Ändert die L/D-Datenstruktur, so dass Information in den Zeilen geordnet ist.}

for $i = 1$ to n **do**
  $c\ell(i) := 0$
end for
for $k = 1$ to $nnz(L)$ **do** {Länge der Spalten bestimmen}
  $c\ell(cn(k)) := c\ell(cn(k)) + 1$
end for
$cst(1) := 1$
for $j = 2$ to n **do**
  $cst(j) := cst(j-1) + c\ell(j-1)$
end for
for $i = 1$ to n **do**
  $cp(i) := cst(i)$ {Anfangszeiger}
end for
for $i = 1$ to n **do** {alle Zeilen}
  for $k = rst(i)$ to $rst(i) + r\ell(i) - 1$ **do**
    $rn(cp(cn(k))) := i$
    $cp(cn(k))$++
  end for
end for
{jetzt enthält $rn(cst(i) : cst(i)+c\ell(i)-1)$ Zeilennummern der i-ten Spalte, und zwar sortiert! }

(A) ——————

for $j = 1$ to n **do** {Anfangszeiger}
  $rp(j) := rst(j)$
end for
for $j = 1$ to n **do** {alle Spalten}
  for $k = cst(j)$ to $cst(j) + c\ell(j) - 1$ **do**
    $cn(rp(rn(k))) := j$
    $rp(rn(k))$++
  end for
end for
{jetzt ist also auch $cn(rst(i) : rst(i) + r\ell(i) - 1)$ sortiert.}

2.2.2 Bemerkung
Bis zur Marke (A) wird die Datenstruktur für die Spalten von L, also die Zeilen von L^T aufgebaut. Danach wird dieser Prozess wiederholt, d.h. die Spaltenstruktur von L^T = Zeilenstruktur von $(L^T)^T = L$ wird aufgebaut.

2.2.3 Bemerkung
Der Aufwand von Algorithmus 2.2.1 ist

$$\mathcal{O}(n) + \mathcal{O}(nnz(L)).$$

Übertragung von A in die (geordnete) L/D-Datenstruktur. Dabei ist

$$struct(L) \supseteq struct(\text{striktes unteres Dreieck von } A).$$

$rst, r\ell, cn, va\ell$ beziehen sich auf L, $rsta, r\ell a, cna$, auf A.

2.2.4 Algorithmus

{Überträgt A in die L/D-Datenstruktur}

for $i = 1$ to n **do**
  scatter $A_{i\cdot}$ in vollen Vektor w
  $d_i := w_i$
  for $k = rst(i)$ to $rst(i) + r\ell(i) - 1$ **do**
    $va\ell(k) = w_{cn(k)}$
  end for
  for $k = rsta(i)$ to $rsta(i) + r\ell a(i) - 1$ **do**
    $w_{cna(k)} := 0$
  end for
end for

Aufwand: $\mathcal{O}(nnz(L) + nnz(A))$.

Der Vektor w ist auf 0 zu initialisieren.

Jetzt können wir die numerische Phase für die Cholesky-Zerlegung formulieren in Anlehnung an Algorithmus 1.3.15 (Doolittle für Cholesky)

for $i = 1$ to n **do**
  for $j = 1$ to $i - 1$ **do**
    $\ell_{ij} = \left(a_{ij} - \sum\limits_{k=1}^{j-1} \ell_{ik} d_{kk} \ell_{jk} \right) / d_{jj}$

end for
$d_{ii} = a_{ii} - \sum\limits_{k=1}^{i-1} \ell_{ik}^2 d_{kk}$
end for

Bei dünn besetzten Matrizen ist

$$\sum_{k=1}^{j-1} \ell_{ik} d_{kk} \ell_{jk}$$

wie ein Innenprodukt mit gepackten, geordneten Vektoren, siehe Algorithmus 1.1.7.

2.2.5 Algorithmus
(Cholesky-Faktorisierung für dünn besetzte Matrix,
numerische Phase)

for $i = 1$ to n **do**
  for $k = rst(i)$ to $rst(i) + r\ell(i) - 1$ **do** {Elemente von Zeile i}
    $j = cn(k)$ {Spaltenindex des Nichnullelements}
    $j2 = rst(i)$ {jetzt „Innenprodukt“ von Zeilen i und j}
    for $j1 = rst(j)$ to $rst(j) + r\ell(j) - 1$ **do**
      while $(cn(j2) < cn(j1))$ **do**
        $j2{+}{+}$
      end while
      if $cn(j2) = cn(j1)$ **then**
        $val(k) = val(k) - val(j1)val(j2)d_{cn(j1)}$
      end if
    end for
    $val(k) = val(k)/d_j$
  end for
  for $j1 = rst(i)$ to $rst(i) + r\ell(i) - 1$ **do**
    $d_i = d_i - val(j1)^2 d_{cn(j1)}$
  end for
end for

2.2.6 Bemerkung
Der Aufwand für Algorithmus 2.2.5 beträgt

$$\sum_{i=1}^{n} \sum_{j \in struct(L_{i\cdot})} [\mathcal{O}(1) + \mathcal{O}(r\ell(i)) + \mathcal{O}(r\ell(j))].$$

Gilt z.B. $r\ell(i) \leq c$, $i = 1, ..., n$, so erhält man Aufwand $\mathcal{O}(c^2 n)$.

Abschnitt 2.3

Symbolische Phase für die LDL^T-Faktorisierung

Erinnerung an die symbolische Phase:

a) Bestimmung einer geeigneten Permutation

b) Bestimmung der L/D-Datenstruktur

Hier: Teil b); Teil a) wird in Kapitel 3 behandelt.

Wir verwenden nur die strukturelle Information von A, nicht die numerischen Werte.

2.3.1 Definition
Für $j = 1, \dots, n$ sei $r(j)$ die Position des ersten Nicht-Null-Elements nach Position j in Spalte $L_{\cdot j}$. (Wir setzen $r(j) = n + 1$, falls eine solche Position nicht existiert.)

Beispiel:

$$L = \begin{pmatrix} x & & & & & & \\ & x & & & & & \\ & & x & & & & \\ \boxed{\text{x}} & \boxed{\text{x}} & & x & & & \\ & x & \boxed{\text{x}} & \boxed{\text{x}} & x & & \\ & & x & & \boxed{\text{x}} & x & \\ & & & & x & \boxed{\text{x}} & x \end{pmatrix} \in \mathbb{R}^{7\times 7}$$

Es ergibt sich

$$r(1) = 4, r(2) = 4, r(3) = 5, r(4) = 5, r(5) = 6, r(6) = 7, r(7) = 8.$$

2.3.2 Satz
Es gilt für $j = 1, \dots, n$

$$struct(L_{\cdot j}) = struct(A_{\cdot j}(j:n)) \cup \bigcup_{\substack{k=1,\dots,j-1 \\ r(k)=j}} struct(L_{\cdot k}(j+1:n)).$$

Beweis: Wir zeigen zuerst

$$\text{(2.3.1)} \quad struct(L_{\cdot j}) = struct(A_{\cdot j}(j:n)) \cup \bigcup_{\substack{k=1,\ldots,j-1 \\ k \in struct(L_{j\cdot})}} struct(L_{\cdot k}(j+1:n)).$$

Beweis zu (2.3.1):
$j \in struct(L_{\cdot j})$, denn $L_{jj} = 1$.
$j \in struct(A_{\cdot j}(j:n))$, denn A ist spd.
Sei $i \in struct(L_{\cdot j})$, $i \neq j$, also $i > j$.

$$0 \neq \ell_{ij} = a_{ij} - \sum_{k=1}^{j-1} \ell_{ik} d_{kk} \ell_{jk}$$

$$\begin{aligned} \ell_{ij} \neq 0 \quad &\Longleftrightarrow \quad a_{ij} \neq 0 \text{ oder } \exists k \in \{1,\ldots,j-1\} \text{ mit } \ell_{ik} \neq 0 \text{ und } \ell_{jk} \neq 0 \\ &\Longleftrightarrow \quad i \in struct(A_{\cdot j}(j+1:n)) \textbf{ oder} \\ &\qquad\;\; \exists k \in \{1,\ldots,j-1\} \text{ mit } i \in struct(L_{\cdot k}) \text{ und } k \in struct(L_{j\cdot}) \\ &\Longleftrightarrow \quad i \in struct(A_{\cdot j}(j+1:n)) \cup \bigcup_{\substack{k=1,\ldots,j-1 \\ k \in struct(L_{j\cdot})}} struct(L_{\cdot k}) \end{aligned}$$

Dies beweist (2.3.1).

Zum Beweis des Satzes bemerken wir zunächst noch

$$\text{(2.3.2)} \qquad r(k) < \ell \Rightarrow struct(L_{\cdot k}(\ell+1:n)) \subseteq struct(L_{\cdot r(k)}(\ell+1:n)),$$

denn aus (2.3.1) mit $j = r(k)$ folgt

$$struct(L_{\cdot r(k)}) \supseteq struct(L_{\cdot k}(r(k)+1:n)) \text{ , da } k \in struct(L_{r(k)\cdot}).$$

Für ein $k \in \{1,\ldots,j-1\}$, $k \in struct(L_{j\cdot})$ gilt

$$r(k) = j \textbf{ oder } r(k) < j.$$

Per Induktion folgt: Für jedes $k \in \{1,\ldots,j-1\}$, $k \in struct(L_{j\cdot})$ gibt es ein $m \in \mathbb{N}$ mit $j = r^m(k)$ $(= r(r(\ldots(r(k))\ldots)))$ und $r^{\ell-1}(k) < r^\ell(k)$ für $\ell = 1,\ldots,m-1$. Wegen (2.3.2) gilt dann

$$\text{(2.3.3)} \qquad struct(L_{\cdot k}(j+1:n)) \subseteq \ldots \subseteq struct(L_{\cdot r^{m-1}(k)}(j+1:n)).$$

Für $k' = r^{m-1}(k)$ gilt $r(k') = j$. Aus (2.3.1) und (2.3.3) folgt somit

$$struct(L_{\cdot j}) = struct(A_{\cdot j}(j:n)) \cup \bigcup_{\substack{k=1,\ldots,j-1 \\ r(k)=j}} struct(L_{\cdot k}(j+1:n)).$$

$\square$

Aufbauend auf diesem Satz bietet sich folgende Methode zum Aufbau der Datenstruktur für L an:

2.3.3 Algorithmus
(Cholesky-Faktorisierung, symbolische Phase)

for $i = 1$ to n **do**
  initialisiere eine leere Liste $\mathfrak{L}_i$
    $\{$ $\mathfrak{L}_i$ nimmt alle Spalten j von L auf mit $r(j) = i$ $\}$
end for
for $j = 1$ to $n - 1$ **do** $\{$bestimme Spalte $j\}$
  $$struct(L_{\cdot j}) = struct(A_{\cdot j}(j:n)) \cup \bigcup_{k \in \mathfrak{L}_j} struct(L_{\cdot k}(j+1:n))$$
  bestimme $r(j)$
  füge j in $\mathfrak{L}_{r(j)}$ ein
end for

2.3.4 Bemerkung
Die Listen $\mathfrak{L}_k$ vermeiden die sonst aufwändige Suche nach ℓ mit $r(\ell) = k$. Beim gepackten (spaltenweisen!) Format für L geht das einfach mit zwei zusätzlichen Feldern *headers* und *ℓinks* der Länge n.

Beispiel:

$$\begin{bmatrix} x & & & & & \\ & x & & & & \\ & & x & & & \\ x & x & & x & & \\ x & & & x & x & \\ & x & x & x & & x \end{bmatrix}$$

i/k	1	2	3	4	5	6	7	8
$c\ell$	2	2	1	2	0			
cst	1	3	5	6	8			
rn	4	5	4	6	6	5	6	
$headers$	-	-	-	2	4	3	6	
$\ell inks$	0	1	0	0	0	5		

Die Vereinigungsoperation in Schritt j führt man z.B. folgendermaßen durch:
  $\{p$: globaler Zeiger$\}$

$cst(j) := p$; $c\ell(j) := 0$; $r(j) := n + 1$;
scatter $A_{\cdot j}$ in w $\{w$ globaler Vektor$\}$
for all $k \in \mathfrak{L}_j$ **do**
  scatter $L_{\cdot k}$ in w
end for
for $\ell = csta(j)$ to $csta(j) + c\ell a(j) - 1$ **do** $\{$Spalte j von $A\}$
  $i = rna(\ell)$
  if $w_i \neq 0$ **then**
    $w_i := 0$
    if $i > j$ **then** $\{$füge i in Struktur für $L_{\cdot j}$ ein$\}$
      $rn(p) := i$
      $c\ell(j){+}{+}$
      $p{+}{+}$
      if $i < r(j)$ **then**
        $r(j) := i$
      end if
    end if
  end if
end for
for all $k \in \mathfrak{L}_j$ **do** $\{$relevante Spalten von $L\}$
  for $\ell = cst(k)$ to $cst(k) + c\ell(k) - 1$ **do**
    $i = rn(\ell)$
    if $w_i \neq 0$ **then**
      $w_i := 0$
      if $i > j$ **then** $\{$füge i in Struktur für $L_{\cdot j}$ ein$\}$
        $rn(p) := i$
        $c\ell(j){+}{+}$
        $p{+}{+}$
        if $i < r(j)$ **then**
          $r(j) := i$
        end if
      end if
    end if
  end for
end for
if $r(j) < n + 1$ **then**
  füge j in die Liste $\mathfrak{L}_{r(j)}$ ein
end if

Für den Gesamtrechenaufwand erhalten wir

$$\mathcal{O}(n) + \sum_{j=1}^{n-1} \sum_{k \in \mathcal{L}_j} \mathcal{O}(c\ell(k)) + \mathcal{O}(c\ell a(k)).$$

($\mathcal{O}(c\ell a(k))$: k-te Spalte von $A = \mathcal{O}(c\ell(k))$.)
In dieser Doppelsumme kommt jeder Index k allerdings nur einmal vor. Außerdem ist $c\ell a(k) \leq c\ell(k)$. Der Aufwand ist also

$$\mathcal{O}(n) + \sum_{k=1}^{n-1} \mathcal{O}(c\ell(k)).$$

Im Fall $c\ell(k) \leq c$ also $\mathcal{O}(cn)$.

KAPITEL 3

Geeignete Permutationen bei symmetrischer Struktur

Abschnitt 3.1

Grundbegriffe aus der Graphentheorie

3.1.1 Definition
Ein *(ungerichteter) Graph* $G = (V, E)$ ist gegeben durch

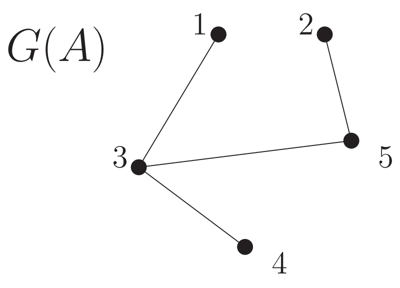
$$\begin{aligned} V = \{1, \ldots, n\} \quad & \text{„}\textit{Knoten}\text{“}, \\ E \subseteq V \times V \quad & \text{„}\textit{Kanten}\text{“}. \end{aligned}$$

3.1.2 Definition
Die Matrix $A \in \mathbb{R}^{n \times n}$ sei symmetrisch. Der zu A gehörige Graph $G(A)$ ist gegeben durch

$$\begin{aligned} V &= \{1, \ldots, n\}, \\ E &= \{\{i, j\} \in V \times V : a_{ij} \neq 0 \wedge i \neq j\}. \end{aligned}$$

3.1.3 Beispiel

$$A = \begin{bmatrix} x & & x & & \\ & x & & & x \\ x & & x & x & x \\ & & x & x & \\ & x & x & & x \end{bmatrix}$$



3.1.4 Definition
$G = (V, E)$ sei ein Graph.

a) $i, j \in V$ heißen *verbindbar*, falls $i = i_0, i_1, \ldots, i_\ell = j \in V$ existieren mit $\{i_\nu, i_{\nu+1}\} \in E$, $\nu = 0, \ldots, \ell - 1$. Dabei heißt $k = (i = i_0, i_1, \ldots, i_\ell = j)$ ein *Kantenzug* der Länge $\ell(k) = \ell$ von i_0 nach i_ℓ.

b) G heißt *zusammenhängend*, falls alle $i \neq j \in V$ verbindbar sind.

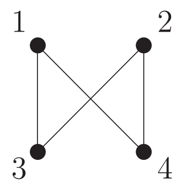
c) G heißt *zyklusfrei*, falls es keinen Kantenzug der Länge $\ell > 2$ gibt, so dass
$$i_\nu \neq i_\mu,\ \nu \neq \mu,\ \nu, \mu = 0, \ldots, \ell - 1,\ i_\ell = i_0.$$

d) Ein zusammenhängender, zyklusfreier Graph heißt *Baum*.

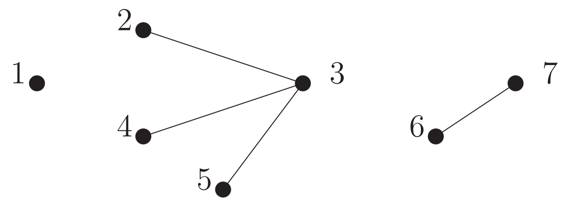
3.1.5 Beispiel

a) $G(A)$ aus Beispiel 3.1.3 ist ein Baum. Es ist $(1, 3, 5, 2)$ ein Kantenzug von 1 nach 2. Auch $(1, 3, 5, 3, 5, 2)$ ist ein solcher.

b) Folgender Graph ist zusammenhängend, aber kein Baum.



c) Folgender Graph ist nicht zusammenhängend, aber zyklusfrei



Die Relation „ist verbindbar mit" wird zu einer Äquivalenzrelation, wenn man per Definition festlegt, dass jeder Knoten mit sich selbst verbindbar ist.

3.1.6 Definition
Sei $G = (V, E)$ ein Graph.

a) $G' = (V', E')$ heißt *Teilgraph* von G, falls $V' \subseteq V$, $E' \subseteq E$.

b) Für $V' \subseteq V$ heißt $G' = (V', E')$ der von V' *induzierte Teilgraph* von G, falls gilt
$$E' = \{e \in E : e = \{i, j\}, i, j \in V'\}.$$

c) Die von den Äquivalenzklassen bezüglich „verbindbar mit“ induzierten Teilgraphen von G heißen *Zusammenhangskomponenten*.

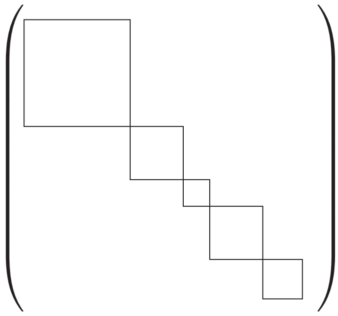
3.1.7 Beispiel
Der Graph aus Beispiel 3.1.5 c) besitzt drei Zusammenhangskomponenten.

3.1.8 Satz
Für $A \in \mathbb{R}^{n\times n}$, $A = A^T$ besitze $G(A)$ die m Zusammenhangskomponenten (V_i, E_i), $i = 1, \dots, m$. Es sei π eine Permutation der Menge $\{1, \dots, n\}$, die nach Zugehörigkeit zu den V_i sortiert, d.h. es gelte

$$V_i = \{\pi(j_i + 1), \dots, \pi(j_{i+1})\} \quad \text{mit } j_i = \sum_{\ell=1}^{i-1} |V_\ell|.$$

Sei P die zugehörige Permutationsmatrix. Dann besitzt die Matrix PAP^T die Blockdiagonalgestalt:

$PAP^T =$



$= \operatorname{diag}(B_1, \dots, B_m)$

mit $B_i \in \mathbb{R}^{|V_i|\times|V_i|}$, $i = 1, \dots, m$.

Beweis: Eigentlich trivial. Für einen formalen Beweis:

$$M_i = \{j_i + 1, \dots, j_{i+1}\} \qquad \text{also } \pi(M_i) = V_i$$

Sind nun i, j so, dass $i \in M_\ell$, $j \in M_k$ mit $\ell \neq k$, so gilt $\pi(i) \in V_\ell$, $\pi(j) \in V_k$. Nach Voraussetzung ist dann $0 = a_{\pi(i),\pi(j)} = (PAP^T)_{ij}$ $\square$

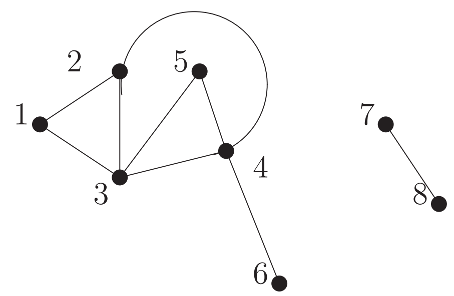
Beim Lösen eines linearen Gleichungssystems kann man sich also auf die einzelnen Diagonalblöcke beschränken.

Wie findet man Zusammenhangskomponenten? $\rightsquigarrow$ Mit den üblichen Techniken zum Durchlaufen eines Graphen.

3.1.9 Algorithmus
(bestimmt alle Zusammenhangskomponenten)

$nr := 0$
for $v = 1$ to n **do** {alle Knoten}
  if v nicht markiert **then**
    $nr{+}{+}$
    markiere v mit nr
    füge v in eine leere Datenstruktur D ein
    while $D \neq \emptyset$ **do**
      entferne Knoten w aus D
      for all Kanten $\{w, u\}$ **do**
        if u noch nicht markiert **then**
          markiere u mit nr
          füge u in D ein
        end if
      end for
    end while
  end if
end for

D Schlange: „breadth first“
D Stapel: „depth first“



D Schlange

D	1	2	3	4	5	6	7	8
$\emptyset$	-	-	-	-	-	-	-	-
1	1							
2,3		1	1					
3,4				1				
4,5					1			
5,6						1		
6								
$\emptyset$								
7							2	
8								2
$\emptyset$								

Der folgende Satz ist aus dem Grundstudium bekannt.

3.1.10 Satz

a) Algorithmus 3.1.9 bestimmt die Zusammenhangskomponenten. Die Zusammenhangskomponente V_i besteht aus den mit i markierten Knoten.

b) Der Aufwand beträgt $\mathcal{O}(|V| + |E|)$, wenn man den Graphen in einer Adjazenz-Listen-Darstellung abspeichert.

c) Unser gepacktes Format (oder Listenformat) ohne $va\ell$ ist gerade eine solche Darstellung für $G(A)$. Wir brauchen also nur ein Feld für die Markierungen und die Datenstruktur D.

Ab jetzt gehen wir davon aus, dass $G(A)$ zusammenhängend ist, d.h. die Zusammenhangskomponenten seien bereits gefunden.

Abschnitt 3.2

Reduktion des Profils

3.2.1 Definition
Für $A = (a_{ij}) \in \mathbb{R}^{n\times n}$, eventuell unsymmetrisch, ist die *Enveloppe* gegeben durch zwei Vektoren p, q mit

$$\begin{aligned} p(i) &= \min_{j=1}^{i}\{j : a_{ij} \neq 0\} && \text{„}\textit{linke Enveloppe}\text{“}, \\ q(j) &= \min_{i=1}^{j}\{i : a_{ij} \neq 0\} && \text{„}\textit{obere Enveloppe}\text{“}. \end{aligned}$$

Es wird $a_{ii} \neq 0$ vorausgesetzt, also ist

$$p(i) \leq i, \qquad q(i) \leq i.$$

Beispiel:

	1	2	3	4	5
1	x	x			x
2	x	x		x	
3		x	x		
4			x	x	
5			x	x	x

$$\begin{aligned} p &= (1, 1, 2, 3, 3) \\ q &= (1, 1, 3, 2, 1) \end{aligned}$$

Bei Matrizen mit symmetrischer Struktur ist $p = q$. Insbesondere gilt dies also auch für symmetrische Matrizen.

3.2.2 Definition

a) Das *Profil* einer Matrix A ist die Menge

$$Pr(A) = \{(i, j) : p(i) \leq j \leq i \text{ oder } q(j) \leq i \leq j\}$$

alias: „*Skyline*“.

Das Profil für obiges Beispiel ist

	1	2	3	4	5
1	x	x			x
2	x	x		x	
3		x	x		
4			x	x	
5			x	x	x

b) $A \in \mathbb{R}^{n \times n}$ heißt *Bandmatrix* mit den *halben Bandbreiten* b_ℓ, b_u, falls b_ℓ, b_u minimal sind mit

$$\begin{aligned} p(i) &\geq \max\{1, i - b_\ell\} \\ q(j) &\geq \max\{1, j - b_u\} \end{aligned}$$

Halbe Bandbreiten für obiges Beispiel:

	1	2	3	4	5
1	x	x	x	x	x
2	x	x		x	
3		x	x		
4			x	x	
5			x	x	x

b_u

b_ℓ

$b_\ell = 2, \; b_u = 4.$

3.2.3 Definition
$A = (a_{ij})$ heißt *block-tridiagonal*, falls es $n_1, \ldots, n_N$,

$$\sum_{i=1}^{N} n_i = n$$

gibt mit

$$A = \begin{bmatrix} B_1 & C_1 & & & \\ A_2 & B_2 & C_2 & & \\ & A_3 & B_3 & C_3 & \\ & & A_4 & B_4 & \ddots \\ & & & \ddots & \ddots \end{bmatrix}$$

mit $B_i \in \mathbb{R}^{n_i \times n_i}$, $A_i \in \mathbb{R}^{n_i \times n_{i-1}}$, $C_i \in \mathbb{R}^{n_i \times n_{i+1}}$. Falls $A = A^T$ ist, gilt $B_i = B_i^T$ und $C_i = A_{i+1}^T$.

Thema dieses Abschnitts: Suche Permutationen so, dass das Profil klein wird (d.h. $p(i)$ möglichst groß). Die Struktur soll also nahe der Diagonalen konzentriert sein. Hintergrund ist der folgende Satz:

3.2.4 Satz
Sei $A = A^T$ mit Enveloppe $p_A (= q_A)$. Dann gilt für L in $A = LDL^T$ (Existenz vorausgesetzt) für die Enveloppen p_L, q_L

$$p_L = p_A, \qquad q_L(j) = j,\ \ j = 1, \ldots, n.$$

Beweis: $q_L(j) = j$ klar wegen unterer Dreiecksgestalt von L. Für $i > j$ gilt:

$$\ell_{ij} = \left(a_{ij} - \sum_{k=1}^{j-1} \ell_{ik} d_{kk} \ell_{jk} \right) / d_{jj} \quad \text{(Algorithmus 1.3.15).}$$

Für festes i bedeutet dies

$$a_{ij} = 0 \wedge \ell_{i1}, \ldots, \ell_{i,j-1} = 0 \Rightarrow \ell_{ij} = 0.$$

Per Induktion über j ergibt sich damit

$$\begin{aligned} j < p_A(i) \quad &\Rightarrow \quad \ell_{ij} = 0 \\ j = p_A(i) \quad &\Rightarrow \quad \ell_{ij} \neq 0 \ (= a_{ij}/d_{jj}). \end{aligned}$$

Also gilt $p_A = p_L$. □

Bemerkung:

a) Der Satz gilt auch, wenn man zufällige Nullen berücksichtigen würde.

b) Sind „innerhalb“ des Profils von A Einträge $= 0$, so sind sie in L im Allgemeinen $\neq 0$.

Siehe Übungsaufgabe 18 für mehr Details.

Ab jetzt sei A symmetrisch oder wenigstens die Struktur. Wir benötigen:

3.2.5 Lemma
Sei π eine Permutation auf $\{1, \ldots, n\}$, $A = A^T \in \mathbb{R}^{n \times n}$, $G(A) = (V, E)$ der zugehörige Graph, P die Permutationsmatrix zu π. Dann ist

$$G(PAP^T) = (\{1, \ldots, n\}, E')$$

mit

$$\{i, j\} \in E' \Longleftrightarrow \{\pi(i), \pi(j)\} \in E.$$

$G(PAP^T)$ entsteht also aus $G(A)$ durch „Umnummerierung“ der Knoten.
Beweis: Trivial. $\square$

3.2.6 Definition
$G = (\{1,\dots,n\}, E)$ sei ein (zusammenhängender) Graph, $S_1 \subseteq \{1,\dots,n\}$ fest. Dann sind (die zu S_1 gehörigen) *Levelmengen* $S_1, S_2, \dots$ definiert durch

$$\begin{aligned} S_2 &= \{v : \exists w \in S_1 \text{ mit } \{v,w\} \in E\} \setminus S_1 \\ S_3 &= \{v : \exists w \in S_2 \text{ mit } \{v,w\} \in E\} \setminus (S_1 \cup S_2) \\ \vdots &= \quad \vdots \\ S_i &= \{v : \exists w \in S_{i-1} \text{ mit } \{v,w\} \in E\} \setminus (S_1 \cup \dots \cup S_{i-1}). \end{aligned}$$

S_i ist also gerade die Menge aller Knoten, die von S_1 durch einen Kantenzug der Länge $i-1$ — und keinen kürzeren — erreichbar sind.

3.2.7 Lemma
Für die Levelmengen S_i gilt:

a) Es existiert ein $i_0 \geq 1$, $i_0 \leq n$, so dass $S_j = \emptyset$ für $j > i_0$.

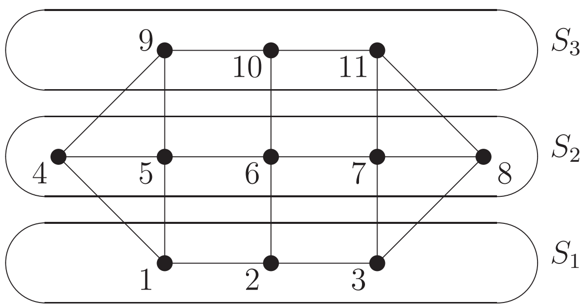
b) $S_i \cap S_j = \emptyset$ für $i \neq j$.

c) $\bigcup_i S_i = \{1,\dots,n\}$ (Voraussetzung: G ist zusammenhängend).

d) $S_i = \{v : \exists w \in S_{i-1} \text{ mit } \{v,w\} \in E\} \setminus (S_{i-1} \cup S_{i-2})$.

Beweis: Trivial. $\square$

3.2.8 Beispiel



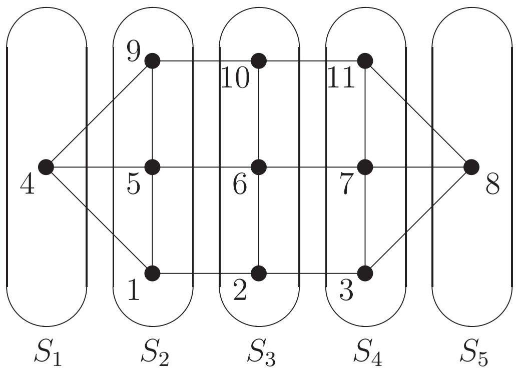
Levelmengen startend mit $S_1 = \{1,2,3\}$:

$$\begin{aligned} S_1 &= \{1,2,3\} \\ S_2 &= \{4,5,6,7,8\} \\ S_3 &= \{9,10,11\}. \end{aligned}$$

Zugehörige Matrix PAP^T:

	1	2	3	4	5	6	7	8	9	10	11
1	x	x		x	x						
2	x	x	x			x					
3		x	x				x	x			
4	x			x	x				x		
5	x			x	x	x			x		
6		x			x	x	x			x	
7			x			x	x	x			x
8			x				x	x			x
9				x	x				x	x	
10						x			x	x	x
11							x	x		x	x

Also $b_u = b_\ell = 5$.



Levelmengen startend mit $S_1 = \{4\}$:

$$
\begin{aligned}
S_1 &= \{4\}\\
S_2 &= \{1,5,9\}\\
S_3 &= \{2,6,10\}\\
S_4 &= \{3,7,11\}\\
S_5 &= \{8\}
\end{aligned}
$$

Zugehörige Matrix PAP^T:

	4	1	5	9	2	6	10	3	7	11	8
4	x	x	x	x							
1	x	x	x		x						
5	x	x	x	x		x					
9	x		x	x			x				
2		x			x	x		x			
6			x		x	x	x		x		
10				x		x	x			x	
3					x			x	x		x
7						x		x	x	x	x
11							x		x	x	x
8								x	x	x	x

Also $b_u = b_\ell = 3$.

Allgemein gilt: Ordnet man die Knoten in der Reihenfolge $S_1, S_2, S_3, \ldots$ an, so besitzt die permutierte Matrix Block-Tridiagonalgestalt mit $n_i = |S_i|$. Insbesondere gilt für die halben Bandbreiten

$$b_\ell = b_u \leq \max_{i=1}^{n-1}\{|S_i| + |S_{i+1}|\} - 1.$$

Der Algorithmus von Cuthill-McKee (1969) verwirklicht eine Nummerierung nach Levelmengen in einer breadth-first-Suche. Er verwendet für S_1 eine einelementige Menge.

3.2.9 Algorithmus
(Cuthill-McKee)
{bestimmt die CM-Anordnung π; Ausgangsmenge $S_1 = \{v\}$}

$\ell := 1$; $s(v) := 1$; $\pi(v) = 1$;
enqueue(S, v) {füge v in die leere *Schlange* S ein}
while $S \neq \emptyset$ **do**
  $w = $ *dequeue*(S) {entferne w aus S}
  for all x mit $\{w, x\} \in E$ **do**
    if x nicht markiert **then**
      ℓ++
      $\pi(x) = \ell$
      $s(x) = s(w) + 1$
      enqueue(S, x) {füge x in die Schlange ein}
    end if
  end for
end while

Beachte: „nicht markiert“ $\widehat{=}$ $\pi(x)$ nicht definiert. $s(w) : w \in S_{s(w)}$

Algorithmus 3.2.9 ist ein Spezialfall von Algorithmus 3.1.9, also:
Aufwand: $\mathcal{O}(|V| + |E|)$.

3.2.10 Lemma

a) Der Algorithmus bestimmt die Levelmengen S_i von $G(A)$ bei Start $S_1 = \{v\}$, wobei $S_i = \{w : s(w) = i\}$.

b) Für die CM-Anordnung π gilt für die permutierte Matrix $B = PAP^T$

   (i) B ist block-tridiagonal.

   (ii) Für die Enveloppe p von B gilt: $p(i) \leq p(i+1), i = 1, \dots, n-1$.

Beweis:

a) (eigentlich Info II)

   Vorüberlegung: ist $S = [x_1, \dots, x_p]$ die Schlange in irgendeinem Moment des Algorithmus, so gilt:

   $$s(x_1) = \ldots = s(x_p) \tag{3.2.1}$$

oder

$$s(x_1) \leq \ldots \leq s(x_p) = s(x_1) + 1 \tag{3.2.2}$$

Beweis dazu: Am Anfang ($S = \{v\}$) ist das trivial. Die *dequeue*-Operation beläßt (3.2.1) bei (3.2.1) oder (3.2.2) bei (3.2.2) oder überführt (3.2.2) nach (3.2.1). Die *enqueue*-Operation fügt nach zuvor erfolgten *dequeue* von x_1 einen Knoten x mit $s(x) = s(x_1) + 1$ am Ende an.

Jetzt zum Beweis:

Per Induktion über i zeigen wir $S_i = \{x : s(x) = i\}$.

$i = 1$: $S_1 = \{v\}$. Einziger Knoten mit $s(x) = 1$ ist gerade v.

$i \rightsquigarrow i+1$: Sei $x \in S_{i+1}$ und $(v, \ldots, \tilde{w}, x)$ ein Kantenzug der Länge $i+1$ von v nach x. Dann gilt $\tilde{w} \in S_i$, also nach Induktionsannahme $s(\tilde{w}) = i$.

Fall 1: x wird markiert bei Entfernung von $\tilde{w}$ aus S. Dann ist $s(x) = s(w) + 1 = i + 1$.

Fall 2: x wurde früher markiert, bei Entfernung von $\hat{w}$. Dann ist $s(\hat{w}) \leq s(\tilde{w})$ wegen (3.2.1), (3.2.2).

Angenommen, $s(\hat{w}) < s(\tilde{w})$. Nach Indunktionsannahme ist dann $\hat{w} \in S_{s(\hat{w})}$ mit $s(\hat{w}) < i$. Es existiert also ein Kantenzug der Länge $s(\hat{w})$ von v nach $\hat{w}$; durch Verlängerung um $\{\hat{w}, x\}$ wird dies ein Kantenzug der Länge $\leq i$ von v nach x. $\rightsquigarrow$ Widerspruch zu $x \in S_{i+1}$.

Also ist $s(\hat{w}) = s(\tilde{w}) = i$ und $s(x) = i + 1$. Damit ist gezeigt: $S_{i+1} \subseteq \{x : s(x) = i + 1\}$. Sei nun $s(x) = i + 1$ und x sei markiert worden von w aus mit $s(w) = i$. Nach Induktionsannahme existiert ein Kantenzug der Länge i von v nach w, also der Länge $i + 1$ von v über w nach x. Da $s(x) = i+1$ gilt nach Induktionsannahme $x \notin (S_1 \cup \ldots \cup S_i)$. Also ist $x \in S_{i+1}$.

b) Nach Konstruktion von π gilt

$$\bar{S}_i = \pi(S_i) = \{n_{i-1} + 1, \ldots, n_i\}, i = 1, \ldots, k$$

$(0 = n_0 < n_1 < \ldots < n_k = n)$. Also besitzt B Tridiagonalgestalt mit Blöcken der Dimension $m_i = n_i - n_{i-1}$. ($A_i = C_{i-1}^T$)

Kein C_i^T besitzt eine Nullzeile, denn jeder Knoten aus $\bar{S}_i$ ist mit einem aus $\bar{S}_{i-1}$ verbunden. Also ist schon klar:

$$n_{i-1} + 1 \leq p(\ell) \leq n_i - 1, \quad n_i + 1 \leq \ell \leq n_{i+1}.$$

Zu zeigen bleibt noch:

$$n_i + 1 \leq \ell < k \leq n_{i+1} \Rightarrow p(\ell) \leq p(k).$$

Es sei $\ell = \pi(\ell')$, $k = \pi(k')$. Wegen $\ell < k$ wurde ℓ' also vor k' markiert. Es ist $p(\ell) = \pi(w)$, wenn ℓ' in Kante $\{w, \ell'\}$ bei der Entnahme von w markiert wurde. Entsprechend ist $p(k) = \pi(\tilde{w})$, wobei $\tilde{w}$ später untersucht wurde als w oder $w = \tilde{w}$, also ist $\pi(\tilde{w}) \geq \pi(w)$.

$\square$

Aufgrund von Lemma 3.2.10 hat das Profil von B „Treppengestalt“:

$$\begin{bmatrix}
x & & & & & & & & \\
x & x & & & & & & & \\
x & 0 & x & & & & & & \\
 & x & 0 & x & & & & & \\
 & x & 0 & 0 & x & & & & \\
 & & & & x & x & & & \\
 & & & & x & 0 & x & & \\
 & & & & & & x & x & \\
 & & & & & & x & 0 & x
\end{bmatrix}$$

Hierbei werden „innere Nullen“ nicht ausgenutzt. Ein „Einrücken“ der Skyline wäre aber durchaus wünschenswert.

Idee: Bei CM hat das Profil, „von unten betrachtet“ durchaus solche Einrükkungen. Wir spiegeln deshalb die Matrix an der Antidiagonalen /, d.h. wir drehen die Ordnung um.

3.2.11 Definition

Die sogenannte *umgekehrte CM-Anordnung* (*RCM = reverse Cuthill-McKee*) ist gegeben durch die Permutation ρ

$$\rho(i) = n + 1 - \pi(i), \quad i = 1, \ldots, n.$$

Bezeichnung:

$$B \;= PAP^T \quad \text{„CM“}$$

$$C \quad = RAR^T \qquad \text{„RCM“}$$

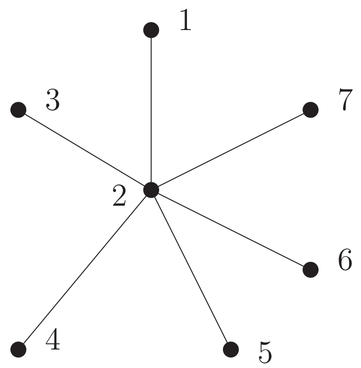
Dann ist

$$C(i, 1:i) = C(1:i, i) = B(n:-1:n+1-i, n+1-i).$$

(i-te Zeile von C (bis Diagonale) $= n+1-i$-te Spalte von B (ab Diagonale).)

3.2.12 Beispiel

$G(A) =$



$S_1 = \{1\}, S_2 = \{2\}, S_3 = \{3, 4, 5, 6, 7\}$
$\Rightarrow$ CM-Anordnung liegt bereits vor.

$$\begin{array}{c|ccccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
1 & x & x & & & & & \\
2 & x & x & x & x & x & x & x \\
3 & & x & x & & & & \\
4 & & x & & x & & & \\
5 & & x & & & x & & \\
6 & & x & & & & x & \\
7 & & x & & & & & x
\end{array}$$

RCM-Anordnung:

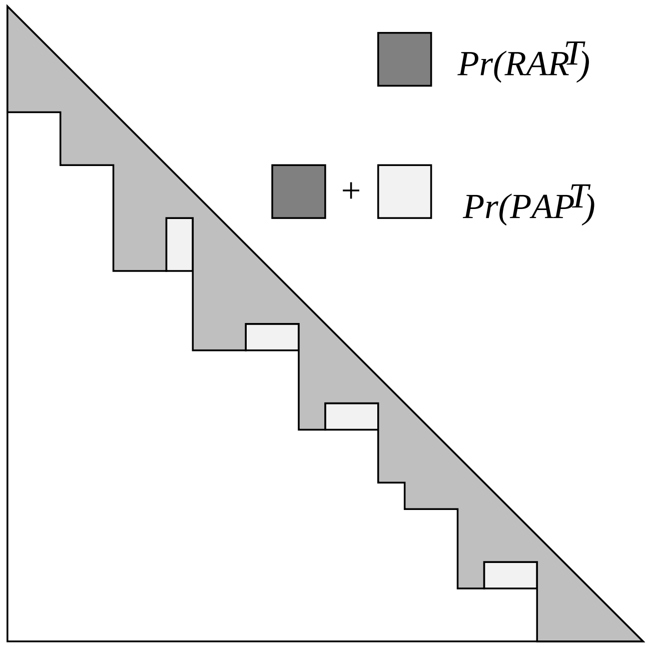
$$\begin{array}{c|ccccccc}
 & 7 & 6 & 5 & 4 & 3 & 2 & 1 \\
7 & x & & & & & x & \\
6 & & x & & & & x & \\
5 & & & x & & & x & \\
4 & & & & x & & x & \\
3 & & & & & x & x & \\
2 & x & x & x & x & x & x & x \\
1 & & & & & & x & x
\end{array}$$

RCM hat sich in der Praxis (insbesondere bei finiten Elementen) gut bewährt und ist stets besser als CM in folgendem Sinne:

3.2.13 Satz
Sei $A = A^T \in \mathbb{R}^{n \times n}$, P sei CM-Permutationsmatrix, R sei RCM-Permutationsmatrix. Dann gilt für die Profile der permutierten Matrizen

$$|Pr(RAR^T)| \leq |Pr(PAP^T)|.$$

Beweis: Formal eher abstoßend; im Prinzip aber klar am Bild:



□

CM bzw. RCM hängen von der Wahl des Startknotens v mit $S_1 = \{v\}$ ab. Gute Wahl dann, wenn $|S_i|$ klein ist für alle i (denn dann ist $p(\ell)$ groß). Es ist schwer, v mit Blick auf $|S_i|$ optimal zu wählen. Stattdessen versuchen wir, die Zahl der Levelmengen möglichst groß zu machen.

3.2.14 Definition
Sei $G = (V, E)$ ein zusammenhängender Graph.

a) Ein Kantenzug in G heißt *Pfad*, wenn der Kantenzug keinen Knoten mehrfach enthält. Die *Länge* eines Kantenzuges k bezeichnen wir mit $\ell(k)$.

b) Für $v, w \in V$ ist der *Abstand* $d(v, w)$ definiert als

$$d(v, w) \quad = \quad \min\{\ell(k) : k \text{ ist Kantenzug, der } v \text{ mit } w \text{ verbindet}\}$$

$$= \min\{\ell(k) : k \text{ ist Pfad, der } v \text{ mit } w \text{ verbindet}\}.$$

c) Die *Exzentrizität* eines Knotens $v \in V$ ist gegeben durch

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

d) *Der Durchmesser* d von G ist gegeben durch

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

3.2.15 Lemma
$G = (V, E)$ sei zusammenhängend. Für die von $S_1 = \{v\}$ induzierten Levelmengen gilt

(i) $S_i \neq \emptyset, \qquad i = 1, \ldots, \epsilon(v) + 1.$

(ii) $S_i = \emptyset, \qquad i > \epsilon(v) + 1.$

Beweis: Trivial. $\square$

3.2.16 Definition
Sei $G = (V, E)$ zusammenhängend. Dann heißt $v \in V$

(i) *peripher*, wenn $\epsilon(v) = d(G)$.

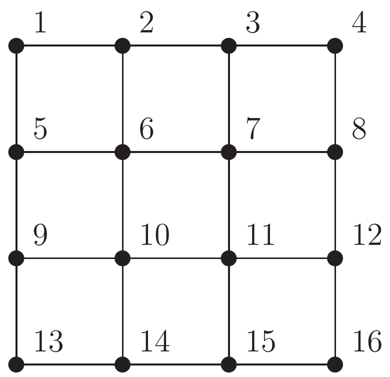
(ii) *pseudoperipher*, wenn gilt

$$d(u, v) = \epsilon(v) \Rightarrow \epsilon(u) = \epsilon(v).$$

Ist v peripher und $d(u, v) = \epsilon(v) = d(G)$, so folgt $\epsilon(u) \geq \epsilon(v) = d(G)$ wie auch $\epsilon(u) \leq d(G)$. Also ist $\epsilon(u) = \epsilon(v) = d(G)$, d.h. periphere Knoten sind pseudoperipher. Pseudoperiphere und periphere Knoten liegen „weit außen“ im Graphen. Pseudoperiphere Knoten sind einfach zu bestimmen ($\rightsquigarrow$ Algorithmus 3.2.18).

3.2.17 Beispiel

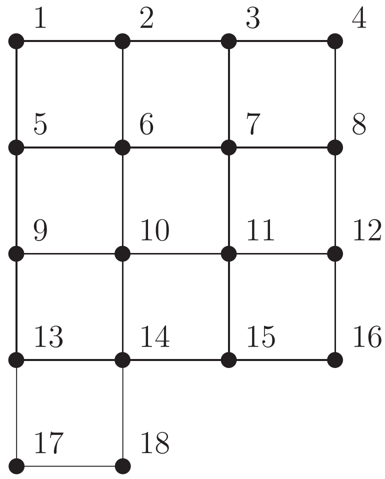
a)



peripher: 1, 4, 13, 16; $d(G) = 6$

pseudoperipher: 1, 4, 13, 16

b)



peripher: 4,17; $d(G) = 7$

pseudoperipher: 4, 17 und 1, 16

Wie bestimmt man einen pseudoperipheren Knoten? Wir berechnen iterativ Knoten v_i mit:

$$v_i \text{ nicht pseudoperipher} \Rightarrow \epsilon(v_{i+1}) > \epsilon(v_i).$$

Ein solches Verfahren stoppt nach spätestens $d(G)$ Iterationen.

3.2.18 Algorithmus
(Gibbs, Pool, Stockmeyer; 1976)

{Bestimmt einen pseudoperipheren Knoten}

wähle v beliebig
$T = \{v\}, m = 0$
while $T \neq \emptyset$ **do**
  entferne w aus T
  $S_1 = \{w\}$
  bestimme $\epsilon(w)$ durch Berechnung aller Levelmengen (Algorithmus 3.2.9)
  if $\epsilon(w) > m$ **then**
    setze $T = S_{\epsilon(w)+1}$ („letzte“ Levelmenge)
    $m = \epsilon(w)$
    $v = w$
  end if
end while

3.2.19 Satz
Algorithmus 3.2.18 bestimmt einen pseudoperipheren Knoten v mit $\epsilon(v) = m$.

Beweis: Der Algorithmus bricht ab, da in T nur eingefügt wird, wenn sich $\epsilon(w)$ erhöht. Das geschieht höchstens $d(G)$ mal. Als v bestimmt wurde, hat sich $\epsilon(w)$ letztmalig erhöht. Für alle $u \in S_{\epsilon(w)+1}$ ist also $\epsilon(u) \leq \epsilon(w)$. Also sogar $\epsilon(u) = \epsilon(w)$. $u \in S_{\epsilon(w)+1}$ ist aber äquivalent zu $d(u,w) = \epsilon(u)$. Also gilt $\epsilon(u) = d(u,w)$, also $\epsilon(u) = \epsilon(w)$. □

Vorbereitend zur nächsten Methode:

3.2.20 Definition
$G = (V, E)$ sei ein Graph.

a) Der Knoten v heißt *inzident* mit Kante e, falls $e = \{v, w\}$, $w \in V$.

b) Der *Grad* $\deg(v)$ ist

$$\deg(v) = |\{e \in E : v \text{ ist inzident mit } e\}|$$

(Anzahl der Kanten, auf denen v liegt).

c) Für $W \subseteq V$ ist *adj*(W), die *Adjazenzmenge* von W, die Levelmenge S_2 bezüglich W (also alle Knoten, die über eine Kante mit einem $v \in W$ verbunden sind und nicht in W liegen).

Wir notieren $adj_G(W)$, wenn der Verweis auf den zugehörigen Graphen wichtig ist.

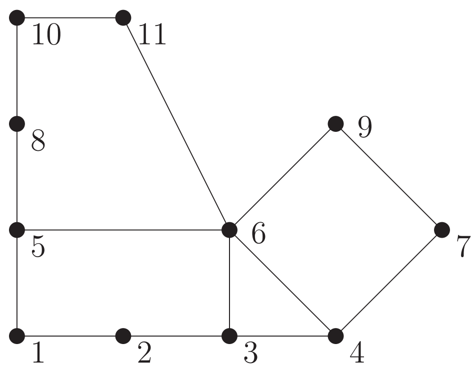
Der Algorithmus von King versucht, das Profil klein zu halten, aber nicht unbedingt die Bandbreite.

3.2.21 Algorithmus
(King, 1970)

{Bestimmt die King-Reihenfolge}

wähle Knoten v mit minimalem Grad.
$A = \{v\}, B = adj(A), C = V \setminus (A \cup B)$
$\pi(v) = 1, \ell = 1$
while $A \neq V$ **do**
  wähle w aus B mit minimalem Grad in $G|_{B \cup C}$ (induzierter Teilgraph)
  $\ell{+}{+}$
  $\pi(w) = \ell$
  $A = A \cup \{w\}, B = adj(A), C = V \setminus (A \cup B)$
end while

3.2.22 Beispiel



King-Reihenfolge: Start mit Knoten vom Grad 2, z.B. 1, 2, 7, 8, 9, 10, 11. Nehme 1:

1 2 5 8 10 11 3 6 4 7 9

	1	2	5	8	10	11	3	6	4	7	9
1	x										
2	x	x									
5	x		x								
8			x	x							
10				x	x						
11					x	x					
3		x					x				
6			x			x	x	x			
4							x	x	x		
7									x	x	
9								x		x	x

Profil $\widehat{=}$ 33 Elemente

CM-Anordnung mit Start 10:

10 8 11 5 6 1 9 3 4 2 7.

RCM-permutierte Matrix:

	7	2	4	3	9	1	6	5	11	8	10
7	x										
2		x									
4	x		x								
3		x	x	x							
9	x				x						
1		x				x					
6			x	x	x		x				
5						x	x	x			
11							x		x		
8								x		x	
10								x	x	x	x

Profil $\widehat{=}$ 36 Elemente

Abschnitt 3.3

Die Minimum-Degree (MD)-Anordnung

Sei $A \in \mathbb{R}^{n \times n}$ symmetrisch (bzw. besitze wenigstens symmetrische Struktur). Wir betrachten jetzt eine Variante der Gauß-Elimination, bei der die Vertauschungen (Pivotwahl) implizit durchgeführt werden. Im Schritt k wird die Pivotposition π_k bestimmt; es sei

$$S_k = \{\pi_1, \ldots, \pi_{k-1}\},\ T_k = \{1, \ldots, n\} \setminus S_k.$$

Die Berechnung von $A^{(k+1)}$ in Schritt k ist wie folgt zu formulieren:

3.3.1 Algorithmus

for $i \in T_{k+1}$ **do**
  $\ell_{ik} = a^{(k)}_{i\pi_k} / a^{(k)}_{\pi_k \pi_k}$
end for
for $i, j \in T_{k+1}$ **do**
  $a^{(k+1)}_{ij} = a^{(k)}_{ij} - \ell_{ik} a^{(k)}_{\pi_k j}$
end for

3.3.2 Definition
Zu $A \in \mathbb{R}^{n \times n}$ mit Permutation $\pi = (\pi_1, \ldots, \pi_n)$ gehört die Folge von *Eliminationsgraphen*

$$G_k = G(A^{(k)}) = (T_k, E_k)$$

mit

$$\begin{aligned} T_k &= \{1, \ldots, n\} \setminus S_k, S_k = \{\pi_1, \ldots, \pi_{k-1}\}, \\ E_k &= \{e = \{i, j\} : a^{(k)}_{ij} \neq 0, i, j \in T_k\}. \end{aligned}$$

3.3.3 Satz
Es gilt

$$\begin{aligned} E_{k+1} = & \quad \{e = \{i, j\} \in E_k : i, j \in T_{k+1}\} \\ & \cup \{e = \{i, j\} : i \neq j \wedge i, j \in adj_{G_k}(\pi_k)\}, \end{aligned}$$

d.h. „alte Kanten“ $\cup$ „entstehende Kanten“, wenn man zufällige Nullen ausschließt.

Beweis: Per Induktion über k; $k = 1$ genügt. Nach Algorithmus 3.3.1 ist klar:

$$\begin{aligned} & \{i,j\} \in E_2 \Longleftrightarrow \{i,j\} \in E_1 \text{ (oder } \{i,\pi_1\} \in E_1 \text{ und } \{\pi_1, j\} \in E_1) \\ \Longleftrightarrow \quad & \{i,j\} \in E_1 \text{ oder } i,j \in adj_{G_1}(\pi_1). \end{aligned}$$

$\square$

Für dünn besetzte Matrizen ist Algorithmus 3.3.1 äquivalent zu:

$\vdots$

for $i, j \in adj_{G_k}(\pi_k)$ **do**
  $a_{ij}^{(k+1)} = a_{ij}^{(k)} - \ell_{ik} a_{\pi_k j}^{(k)}$
end for

Der Rechenaufwand für Schritt k wird also minimal, wenn π_k so gewählt wird, dass $|adj_{G_k}(\pi_k)| = \deg_{G_k}(\pi_k)$ minimal ist.

3.3.4 Definition
Die *Minimum-Degree* (MD)-**Anordnung** ist eine Anordnung $\pi_1, \dots, \pi_n$, für die gilt

$$\deg_{G_k}(\pi_k) = \min_{w \in T_k} \deg_{G_k}(w).$$

Die MD-Anordnung ist im Allgemeinen nicht eindeutig. „Greedy"-Strategie zur Verringerung des Rechenaufwandes.

Mit $F(A)$, dem *Fill-in*, bezeichnen wir die Menge der neuen Kanten in dem Eliminationsgraphen

$$F(A) = (E_2 \cup \ldots \cup E_n) \setminus E_1.$$

Nach Satz 3.3.3 gilt für jede Anordnung

$$|E_{k+1} \setminus E_k| \leq \binom{d_k}{2},$$

wobei $d_k = \deg_{G_k}(\pi_k)$.

In diesem Sinne ist die MD-Strategie auch eine Greedy-Strategie zur Minimierung des Fill-in.

3.3.5 Satz
$G(A)$ sei ein Baum. Bei MD-Anordnung entsteht dann keinerlei Fill-in. Genauer gilt

$$G_k = G|_{T_k} \qquad (3.3.1)$$

und G_k ist ein Baum.

Beweis: Es genügt der Beweis für $k = 2$. Knoten mit minimalem Grad in einem Baum sind die Blätter w, $\deg(w) = 1$. Bei MD-Anordnung ist also $\deg(\pi_1) = 1$. Nach Satz 3.3.3 ist

$$\begin{aligned} E_2 &= \{e \in E_1 : e = \{i,j\}, i,j \neq \pi_1\} \cup \{\{i,j\} : i \neq j, i,j \in T_2, i,j \in adj_{G_1}(\pi_1)\} \\ &= \{e \in E_1 : e = \{i,j\}, i,j \neq \pi_1\} \cup \emptyset, \end{aligned}$$

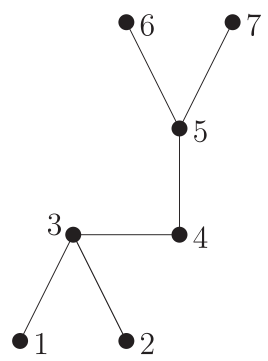
da $|adj_{G_1}(\pi_1)| = 1$. Damit gilt $E_2 \subseteq E_1$ und $G|_{T_2}$ hat keinen Zyklus. Zu zeigen bleibt, dass $G|_{T_2} = G_2$ zusammenhängend ist. Dies folgt aus dem nächstem Satz. □

3.3.6 Satz
Es sei $G(A)$ zusammenhängend. Dann ist auch G_k für jede Pivotwahl zusammenhängend.

Beweis: Wieder reicht $k = 2$. Sei $p = (i_0 = i, i_1, \ldots, i_{\ell-1}, i_\ell = j)$ ein Pfad welcher $i, j \in T_2$ in G_1 verbindet, $i \neq j$.
Fall 1: $i_\nu \neq \pi_1$ für $\nu = 1, \ldots, \ell - 1 \Rightarrow p$ ist ein Pfad in G_2.
Fall 2: $i_\nu = \pi_1$ für genau ein $\nu \in \{1, \ldots, \ell-1\}$. Dann ist $\{i_{\nu-1}, i_\nu\}, \{i_\nu, i_{\nu+1}\} \in E_1$, also nach Satz 3.3.3 $\{i_{\nu-1}, i_{\nu+1}\} \in E_2$, also ist $p' = \{i_0, \ldots, i_{\nu-1}, i_{\nu+1}, \ldots, i_\ell\}$ ein Kantenzug, der $i = i_0$ mit $j = i_\ell$ in G_2 verbindet. □

3.3.7 Beispiel

Baum:



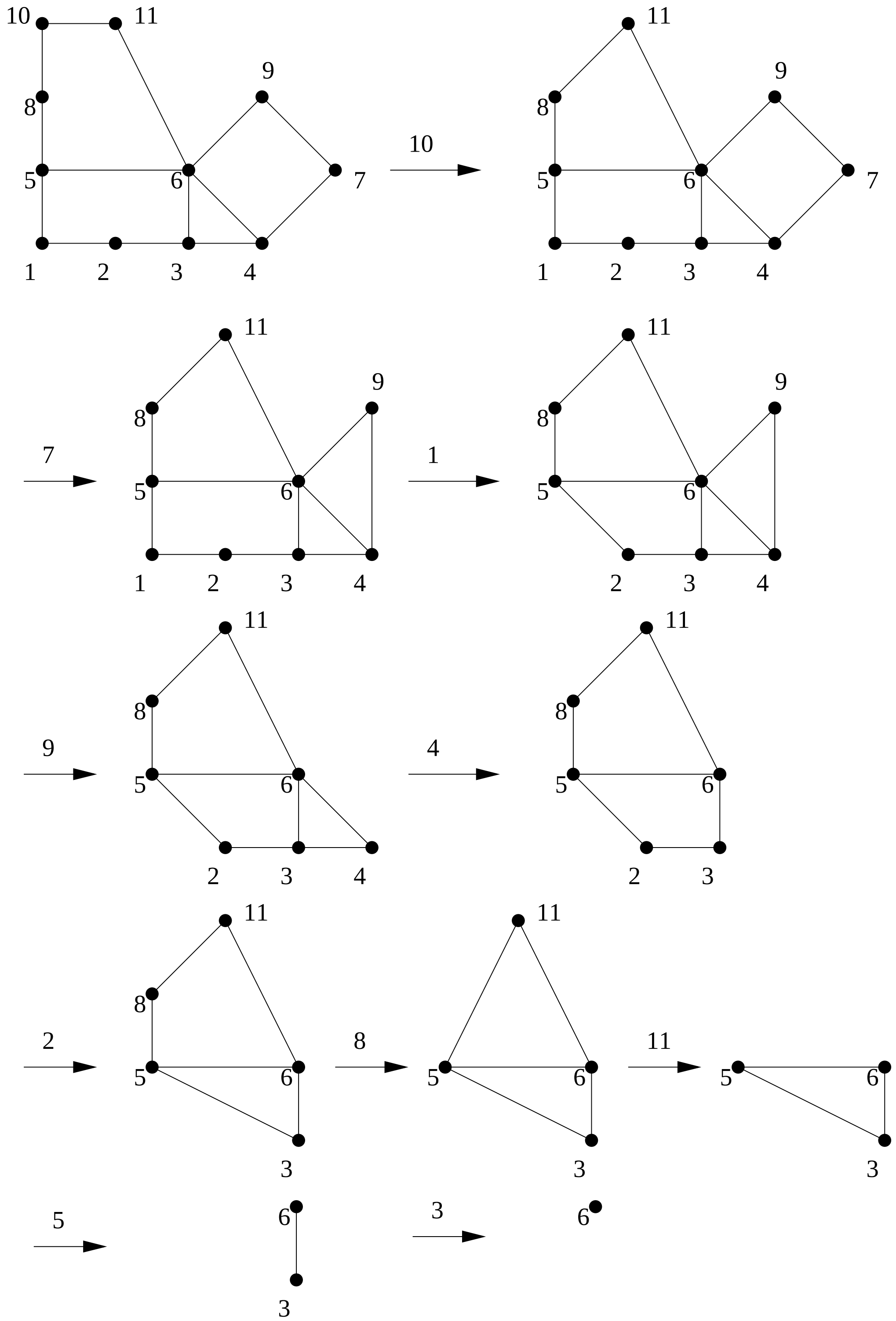
Knoten in MD-Anordnung: 1 ,2, 3, 4, 6, 7, 5

Profil bei MD-Anordnung

$$
\begin{array}{ccccccc}
x & & & & & & \\
 & x & & & & & \\
x & x & x & & & & \\
 & & x & x & & & \\
 & & & & x & & \\
 & & & & & x & \\
 & & & x & x & x & x
\end{array}
$$

3.3.8 Beispiel

MD-Anordnung für Beispiel 3.2.21



	10	7	1	9	4	2	8	11	5	3	6
10	x										
7		x									
1			x								
9		x		x							
4		x		$\otimes$	x						
2			x			x					
8	x						x				
11	x						$\otimes$	x			
5			x			$\otimes$	x	$\otimes$	x		
3					x	x			$\otimes$	x	
6				x	x			x	x	x	x

$\otimes$: Fill-in-Positionen.

Im Rest dieses Abschnitts behandeln wir effiziente Algorithmen und Datenstrukturen zur Berechnung der MD-Anordnung.

Prototyp:

for $k = 1$ to $n-1$ **do**
  bestimme Knoten π_k mit minimalem Grad in G_k.
  bestimme G_{k+1} (d.h. E_{k+1}, z.B. nach Satz 3.3.3)
end for

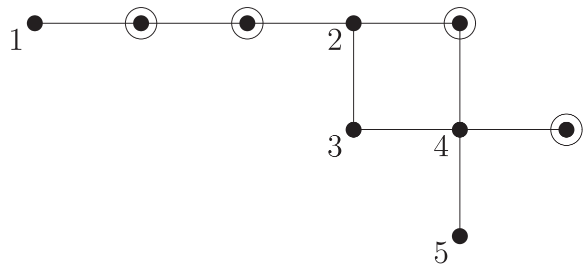
Die Bestimmung von G_{k+1} aus G_k über Satz 3.3.3 ist zu rechen- und vor allem speicheraufwändig. Deshalb stellen wir nun zuerst Rüstzeug für raffiniertere Algorithmen zur Verfügung.
Zitat (aus dem Buch von Duff, Erisman, Reid): Verbesserungen bei der MD-Berechnung führten 1970-1980 zu Beschleunigungen um Faktor 100.

3.3.9 Definition
$G = (V, E)$ sei ein Graph, $S \subseteq V$, $x \in V \setminus S$, $x \neq y \in V$. Dann heißt x *von y über S erreichbar*, wenn ein Kantenzug $(y, x_1, \ldots, x_{\ell-1}, x)$ existiert mit $x_1, \ldots, x_{\ell-1} \in S$. Die von y über S *erreichbare Menge* („reachable set“) ist

$$Reach(y, S) = \{x \in V \setminus S : x \text{ ist von } y \text{ über } S \text{ erreichbar}\}.$$

Beachte: $Reach(y, S) \supseteq adj(y) \setminus S \qquad (\widehat{=}\, \ell = 1)$.

3.3.10 Beispiel



Die mit Doppelkreis dargestellten Knoten entsprechen S.

$$\begin{aligned} Reach(1,S) &= \{2\}, & Reach(2,S) &= \{1,3,4\},\\ Reach(3,S) &= \{2,4\}, & Reach(4,S) &= \{2,3,5\},\\ Reach(5,S) &= \{4\}. \end{aligned}$$

Mit dem Reach-Operator kann man G_k über G_1 charakterisieren:

3.3.11 Satz
Seien $\pi_1, \pi_2, \ldots$ die Pivotknoten, $G_1, G_2, \ldots$ die zugehörigen Eliminationsgraphen $G_k = (T_k, E_k)$; $T_k = \{1, \ldots, n\} \setminus S_k$, $S_k = \{\pi_1, \ldots, \pi_{k-1}\}$. Dann gilt für $y \in T_k$

$$adj_{G_k}(y) = Reach_{G_1}(y, S_k).$$

Beweis: $k = 1 : S_1 = \emptyset$

$$Reach_{G_1}(y, \emptyset) = adj_{G_1}(y) \text{ nach Definition}$$

$k \rightsquigarrow k+1$: Sei $y \in T_{k+1}, x \in T_{k+1}$

$$\begin{aligned} & x \in adj_{G_{k+1}}(y)\\ \iff & x \in adj_{G_k}(y) \text{ oder } x, y \in adj_{G_k}(\pi_k)\\ \iff & x \in Reach_{G_1}(y, S_k) \text{ oder } x, y \in Reach_{G_1}(\pi_k, S_k)\\ \iff & x \text{ ist von } y \text{ erreichbar über } S_k \text{ oder } x \text{ und } y \text{ sind von } \pi_k \text{ erreichbar}\\ & \hfill \text{über } S_k\\ \iff & x \text{ ist von } y \text{ erreichbar über } S_k \text{ oder } x \text{ ist von } y \text{ erreichbar über } S_{k+1}\\ \iff & x \text{ ist von } y \text{ erreichbar über } S_{k+1} \text{ also } x \in Reach_{G_1}(y, S_{k+1}). \end{aligned}$$

□

Mit dem Reach-Operator haben wir eine implizite (im Vergleich zu den Eliminationsgraphen) Darstellung der Gauß-Elimination erreicht. Im Prinzip ist so die MD-Anordnung ohne zusätzlichen Speicheraufwand bestimmbar, aber immer noch mit zu hohem Rechenaufwand. $\rightsquigarrow$ neues Konzept zur Bestimmung von Reach.

3.3.12 Definition
$G = (V, E)$ sei ein Graph und $\mathcal{V} = \{V_1, \ldots, V_m\}$ eine Partition von V, d.h.

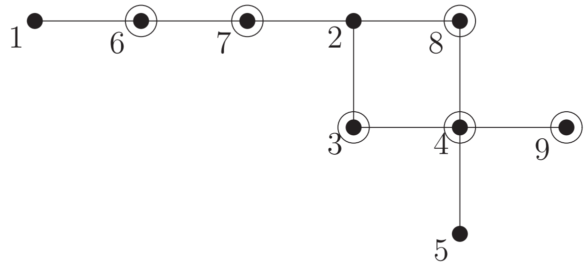
$$\emptyset \neq V_i, \ \bigcup_{i=1}^{m} V_i = V, \ V_i \cap V_j = \emptyset \text{ für } i \neq j.$$

Dann ist der *Quotientengraph* $\mathcal{G} = G/\mathcal{V}$ gegeben durch $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ mit

$$\{V_i, V_j\} \in \mathcal{E} \iff \exists\ v_i \in V_i, v_j \in V_j \text{ mit } \{v_i, v_j\} \in E$$

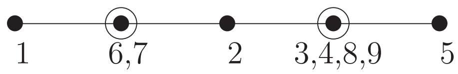
Statt mit $V_1, \ldots, V_m$ kann man die Knoten in $\mathcal{G}$ mit $1, \ldots, m$ bezeichnen oder aber auch durch ausgewählte $v_i \in V_i$.
Zusätzliche Bezeichnung: Für $u \in V$ ist $[u]$ die Menge $V_i \in \mathcal{V}$ mit $u \in V_i$. (Äquivalenzklasse bezüglich der Relation „u ist in derselben Menge V_i wie v“.).

3.3.13 Beispiel



$$\mathcal{V} = \{\{1\}, \{6, 7\}, \{2\}, \{3, 4, 8, 9\}, \{5\}\}$$

$\mathcal{G} = G/\mathcal{V}$:

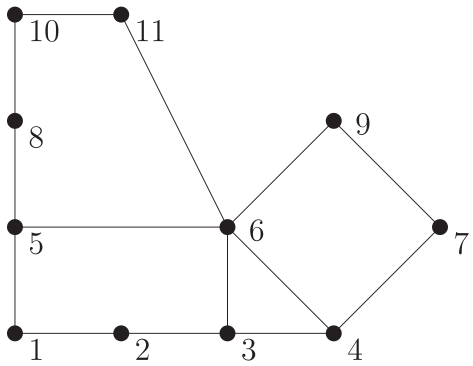


Knoten in $\mathcal{G}$, die mehreren Knoten aus G entsprechen, nennen wir *Superknoten.*

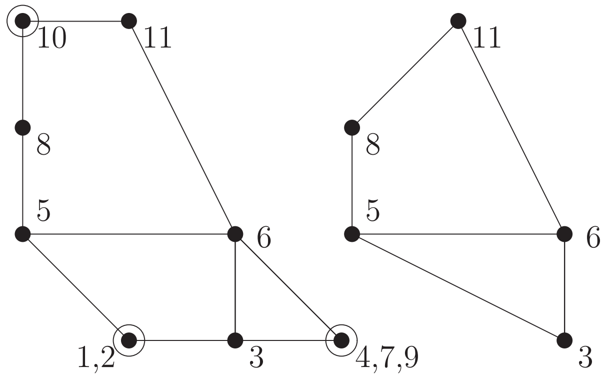
3.3.14 Definition
Sei $G = (V, E)$ ein Graph, $\pi_1, \ldots, \pi_n$ Pivotknoten, $S_k = \{\pi_1, \ldots, \pi_{k-1}\}$, $T_k = \{1, \ldots, n\} \setminus S_k$. Die *Quotienten-Eliminationsgraphen* $\mathcal{G}_k$ sind definiert wie folgt:

$$\begin{aligned} \mathcal{V}_k &= \text{Zusammenhangskomponenten von } G|_{S_k} \cup \{\{w\} : w \in T_k\}, \\ \mathcal{G}_k &= G/\mathcal{V}_k. \end{aligned}$$

3.3.15 Beispiel
G aus Beispiel 3.3.8.



$k = 7$; $S_7 = \{10, 7, 1, 9, 4, 2\}$
$\mathcal{G}_7$:



Vergleich mit G_7 aus Beispiel 3.3.8: Kanten in G_7 entsprechen Kanten in $\mathcal{G}_7$ zwischen normalen Knoten oder Kantenzug der Länge 2 über Superknoten.

Bezeichnung: $\mathcal{C}(S_k)$: Menge der Zusammenhangskomponenten (nur Knotenmenge) von $G|_{S_k}$.

3.3.16 Satz
Für $y \in T_k$ gilt

$$Reach_{G_1}(y, S_k) = \bigcup_{[x] \in Reach_{\mathcal{G}_k}([y], \mathcal{C}(S_k))} [x]$$

oder, kürzer, in falscher Notation aber verständlich (wir identifizieren Mengen von Mengen mit der Menge aus allen Elementen!)

$$Reach_{G_1}(y, S_k) = Reach_{\mathcal{G}_k}([y], \mathcal{C}(S_k)).$$

Beweis:

„$\subseteq$“: Sei $u \in Reach_{G_1}(y, S_k)$.

1. Fall: $u \in adj_{G_1}(y)$. Wegen $y \notin S_k$ ist $[y] = \{y\}$. Es ist also $[u] \neq [y]$.

$$\begin{aligned} u \in adj_{G_1}(y) \quad &\Rightarrow \quad [u] = [y] \text{ (wurde gerade ausgeschlossen)} \\ &\qquad\qquad \text{oder } [u] \in adj_{\mathcal{G}_k}([y]) \\ &\Rightarrow \quad [u] \in Reach_{\mathcal{G}_k}([y], \mathcal{C}(S_k)). \end{aligned}$$

2. Fall: $u \notin adj_{G_1}(y)$.

$$\begin{aligned} u \notin adj_{G_1}(y) \quad &\Rightarrow \quad \exists \text{ Kantenzug } (y, x_1, \ldots, x_{\ell-1}, u) \text{ in } G_1 \\ &\qquad\qquad \text{mit } x_1, \ldots, x_{\ell-1} \in S_k \\ &\Rightarrow \quad x_1, \ldots, x_{\ell-1} \in [x_1] \\ &\Rightarrow \quad \exists \text{ Kantenzug } ([y], [x_1], [u]) \text{ in } \mathcal{G}_k \\ &\Rightarrow \quad [u] \in Reach_{\mathcal{G}_k}([y], \mathcal{C}(S_k)). \end{aligned}$$

„$\supseteq$“: Sei $[u] \in Reach_{\mathcal{G}_k}([y], \mathcal{C}(S_k))$. Sei $([y], [x_1], \ldots, [x_{\ell-1}], [u])$ ein Kantenzug, über den $[y]$ in $\mathcal{C}(S_k)$ erreichbar ist. Da die $[x_\nu]$ zusammenhängend sind, ist $\ell = 1$ oder $\ell = 2$. Im Fall $\ell = 1$ ist $\{y, u\} \in E_1$, im Fall $\ell = 2$ ist u in G_1 über $[x_1]$ erreichbar. In beiden Fällen ist also $u \in Reach_{G_1}(y, S_k)$. Sei $\tilde{u} \in [u], \tilde{u} \neq u$. Dann ist $\tilde{u}$ mit u über einen Kantenzug aus $[u] \subseteq S_k$ verbunden. Also folgt auch $\tilde{u} \in Reach_{G_1}(y, S_k)$.

$\square$

Wichtige Bemerkung: Der Beweis zeigt, dass in $\mathcal{G}_k$ die erreichbaren Knoten immer über Kantenzüge der Länge ≤ 2 erreichbar sind.

3.3.17 Algorithmus
(Bestimmung von $Reach_{\mathcal{G}_k}(y, \mathcal{C}(S_k))$, $y \notin S_k$, grobe Fassung)

$R = \emptyset$
for all $[x] \in adj_{\mathcal{G}_k}([y])$ **do**
  if $[x] \in \mathcal{C}(S_k)$ **then** {Superknoten}
    $R := R \cup adj_{\mathcal{G}_k}([x])$
  else
    $R := R \cup \{x\}$
  end if
end for
$R = R \setminus \{y\}$

Wir wollen im Detail ausarbeiten, wie man die Folge der $\mathcal{G}_k$ algorithmisch bestimmt. Wir weisen zuerst nach, dass man keinerlei zusätzlichen Speicher braucht als für G_1.

3.3.18 Lemma
Sei $G = (V, E)$, $C \subseteq V$ so, dass $G|_C$ zusammenhängend ist. Dann gilt

$$\sum_{x \in C} |adj(x)| \geq |adj(C)| + 2(|C| - 1).$$

Beweis: Die Kantenmenge von $G|_C$ bezeichnen wir mit V_C. In $\sum_{x \in C} |adj(x)|$ wird jede *Kante* aus V_C zweimal gezählt, jede andere Kante (von C nach $V \setminus C$) nur einmal. Also gilt

$$\sum_{x \in C} |adj(x)| = 2|V_C| + |adj(C)|.$$

Weil $G|_C$ zusammenhängend ist, ist $V_C \geq |C| - 1$. $\square$

3.3.19 Lemma
Für $y \in T_k = \{1, \ldots, n\} \setminus S_k$ gilt

$$|adj_{\mathcal{G}_{k-1}}(y)| \geq |adj_{\mathcal{G}_k}(y)|.$$

Beweis: Übungsaufgabe 24.1). $\square$

3.3.20 Satz
Für die Quotienten-Eliminationsgraphen $\mathcal{G}_k = (\mathcal{V}_k, \mathcal{E}_k)$ gilt

$$|\mathcal{E}_{k+1}| \leq |\mathcal{E}_k| \leq |E|, \ k = 1, \ldots, n-1.$$

Beweis: $k = 1$ ist ok, da $\mathcal{G}_1 = G_1$. Die Behauptung gelte für $k - 1$. Sei π_k nächster Pivotknoten.

1. Fall: $\pi_k \notin adj_{\mathcal{G}_k}(\mathcal{C}(S_k))$

$$\Rightarrow \mathcal{C}(S_{k+1}) = \mathcal{C}(S_k) \cup \{\{\pi_k\}\}$$

und $\mathcal{E}_{k+1} = \mathcal{E}_k$.

2. Fall: $\pi_k \in adj_{\mathcal{G}_k}(\mathcal{C}(S_k))$

$$\Rightarrow \mathcal{C}(S_k) = \{[x_1], \ldots, [x_\ell]\}; \ \mathcal{C}(S_{k+1}) = \{[x_1], \ldots, [x_m], [y]\},$$

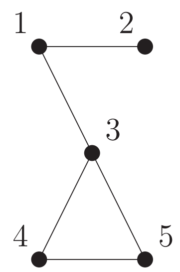
$m < \ell$. ($\pi_k \in adj([x_\nu])$, $\nu = m{+}1, \ldots, \ell$, $[y] = \{\pi_k\} \cup [x_{m+1}] \cup \ldots \cup [x_\ell]$). Nun gilt

$$\begin{aligned}
|\mathcal{E}_k| &= \frac{1}{2} \sum_{[x]\in\mathcal{V}_k} |adj_{\mathcal{G}_k}([x])| \\
&= \frac{1}{2} \sum_{\substack{[x]\in\mathcal{V}_k \\ [x]\neq\pi_k,[x_{m+1}],\ldots,[x_\ell]}} |adj_{\mathcal{G}_k}([x])| + \frac{1}{2} \sum_{\substack{[x]\in\mathcal{V}_k \\ [x]=\pi_k,[x_{m+1}],\ldots,[x_\ell]}} |adj_{\mathcal{G}_k}([x])| \\
&\geq 1.\ \text{Term} + \frac{1}{2} |adj_{\mathcal{G}_k}([y])| \\
&\qquad\qquad (\text{ L.3.3.18 mit } C = \{\pi_k, [x_{m+1}], \ldots, [x_\ell]\}\ ) \\
&\geq \frac{1}{2} \sum_{\substack{[x]\in\mathcal{V}_k \\ [x]\neq\pi_k,[x_{m+1}],\ldots,[x_\ell]}} |adj_{\mathcal{G}_{k+1}}([x])| + \frac{1}{2} |adj_{\mathcal{G}_{k+1}}([y])| \\
&= \frac{1}{2} \sum_{[x]\in\mathcal{V}_{k+1}} |adj_{\mathcal{G}_{k+1}}([x])| = |\mathcal{E}_{k+1}|
\end{aligned}$$

□

Wir benötigen noch eine geeignete Datenstruktur, um (Folge von) Quotienten-Graphen darzustellen. Ähnlich wie beim Listenformat für A stellen wir einen (zunächst gewöhnlichen) Graphen $G = (V, E)$ durch seine Adjazenzlisten-(Kantenlisten)-Darstellung dar.

Felder: *start*, *adj*, *links*



i/k	1	2	3	4	5	6	7	8	9	10
start	1	3	4	7	9					
adj	2	3	1	1	4	5	3	5	3	4
links	2	0	0	5	6	0	8	0	10	0

Wichtige Bemerkung: Im Vergleich zum Listenformat für dünnbesetzte Matrizen haben wir

- kein *val*,

- keine „Diagonale“, d.h. es können leere Listen auftreten, falls G nicht zusammenhängend ist. Dann ist

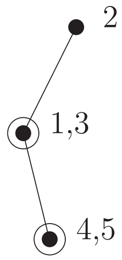
  $$start(v) = 0$$

  zu setzen.

Für alle Quotientengraphen

- repräsentieren wir eine Menge $V_i \subset \mathcal{V}$ durch einen ihrer Knoten. Dazu verwenden wir ein Feld $super$; $super(v) = 1 \Longleftrightarrow v$ repräsentiert ein V_i, sonst $super(v) = 0$,
- verwenden wir adj und $\ell inks$ für die Kantenliste von $\mathcal{G}$, was nach Satz 3.3.20 und Lemma 3.3.19 ausreicht,
- führen wir ein Feld $e\ell im$, welches alle (irgend)einem Superknoten angehörigen Knoten markiert.

Beispiel: Obiges Beispiel mit $\mathcal{V} = \{\{1,3\} \,\widehat{=}\, 1, 2, \{4,5\} \,\widehat{=}\, 4\}$



i/k	1	2	3	4	5	6	7	8	9	10
$start$	1	3	4	7	9					
adj	2	4	1				1			
$\ell inks$	2	0	0				0			
$e\ell im$	1	0	1	1	1					
$super$	1	0	0	1	0					

Bemerkung: Die Adjazenzliste für einen Superknoten kann eventuell den Platz der Adjazenzlisten mehrerer Knoten $\in$ Superknoten beanspruchen. (Im Beispiel nicht der Fall.) Der Platz reicht aber immer aus (siehe Beweis zu Satz 3.3.20).
Nicht mehr verwendete Knoten v ($e\ell im(v) = 1$ aber $super(v) = 0$) werden nicht systematisch entfernt.

Gerüst für einen Algorithmus zur Bestimmung der MD-Anordnung.

3.3.21 Algorithmus
(Bestimmt MD-Anordnung $\pi(1), \pi(2), \ldots, \pi(n)$)

for $i = 1$ to n **do**
  $\deg(i) = |adj(i)|$ {Initialisierung}
end for
$S = \emptyset, T = \{1, \ldots, n\}$
for $i = 1$ to n **do**
  {bestimme Anordnung unter Verwendung von Quotienten-Graphen}
  finde Knoten $\pi(i)$ mit minimalem Grad aus T
  bestimme $R = Reach(\pi(i), \mathcal{C}(S))$
  bilde neuen Superknoten aus $\pi(i)$ und den adjazenten Superknoten
  $S = S \cup \{\pi(i)\},\ T = T \setminus \{\pi(i)\}$
  {datiere $\mathcal{G}$ auf}
  datiere die Adjazenzliste R für Superknoten $\pi(i)$ auf
  for all $w \in R$ **do** {nur hier hat sich $\mathcal{G}$ geändert}
    ersetze in Adjazenzliste für w den ersten eliminierten Knoten y
    mit $super(y) = 0$ durch $\pi(i)$
    datiere $\deg(w)$ auf
  end for
end for

„Definition“: In einem Algorithmus zur Faktorisierung einer dünnbesetzten Matrix ist eine **„$\mathcal{O}(n^2)$-Falle“** ein Teil des Algorithmus mit Aufwand $\Omega(n^2)$.

Hintergrund: Algorithmen sollten Aufwand $\mathcal{O}(n)$ bzw. $\mathcal{O}(\frac{nnz^2}{n})$ besitzen.

3.3.22 Beispiel
Gegeben ist ein Feld $a[1], \ldots, a[n] \in \mathbb{R}$. Gesucht ist $m = \min\limits_{i=1}^{n} a[i]$.
Bekannt: Aufwand ist $\Omega(n)$.

Im MD-Algorithmus (Algorithmus 3.3.21) existiert folgende Schleife

for $i = 1$ to n **do**
  $\vdots$
  (1) finde $m = \min\limits_{w \in T} \deg(w)$
  (2) datiere $\deg(w)$ auf für alle $w \in R$
end for

Ist T gegeben durch $\{1,\dots,n\} \setminus \{\pi(1),\dots,\pi(i-1)\}$, so ergibt sich der Aufwand für (1) zu $\Omega(n-i)$, für Algorithmus 3.3.21 also

$$\sum_{i=1}^{n} \Omega(n-i) = \Omega(n^2) \qquad \leftarrow \mathcal{O}(n^2)\text{-Falle.}$$

Alternativen:

a) Verwende einen Heap. Dann gilt

Aufwand für (1): $\mathcal{O}(1)$

Aufwand für (2): $|R| \cdot \Omega(\log(n-i))$

$\rightsquigarrow$ Gesamtaufwand: $c \cdot \Omega(n \log n)$ (falls $|R| \le c \quad \forall i$).

b) „*Threshold Searching*“

Gegeben: $a[1],\dots,a[n] \in \mathbb{N}$, $\mathit{thresh} \le \min_{j=1}^{n} a[j]$ („=“ sehr wahrscheinlich)

Gesucht: i mit $a[i] = \min_{j=1}^{n} a[j]$.

Prinzip: Berechne i wie bei Beispiel 3.3.22, aber stop, falls Wert *thresh* gefunden wird.

$\mathit{min}1 = a[1]$, $i1 = 1$, $i = 0$
while ($i < n$ und nicht gefunden) **do**
  $i{+}{+}$
  gefunden = ($a[i] == \mathit{thresh}$)
  if $a[i] < \mathit{min}1$ **then**
    $i1 = i$
    $\mathit{min}1 = a[i]$
  end if
end while
{jetzt gilt: gefunden und $a[i] = \mathit{thresh}$ oder $i = n$ und $a[i1] = \min a[j]$}
if nicht gefunden **then**
  $i = i1$
  $\mathit{thresh} = a[i]$ {=Minimum}
end if

Worst case-Abschätzung bei Einbau in MD-Algorithmus wie in den vorliegenden Matlab-Programmen: $\Omega(n^2)$ (thresh jedes Mal zu klein).

Unter folgenden Annahmen ergibt sich ein Aufwand $\mathcal{O}(n)$

(i) thresh habe mit Wahrscheinlichkeit $\geq 1 - \frac{d}{n-i}$ (i-ter Schritt) den „richtigen" Wert (d fester Wert),

(ii) alle vorkommenden Grade sind in jedem Schritt im Intervall $[mindeg, c]$ (c fest) gleichverteilt.

Dann ergibt sich für den Aufwand im Mittel

$$\text{Aufwand } \leq \sum_{i=1}^{n} \frac{d}{n-i}(n-i) + \sum_{i=1}^{n} 1 \cdot \mathcal{O}(c) = \mathcal{O}(n(c+d))$$

c) „Listenverwaltung" (=„Hashing")

Wir legen Listen $\mathcal{L}_i$ an, $i = 0, \ldots, n-1$. Liste $\mathcal{L}_i$ enthält alle Knoten mit Grad i. Es soll mit Aufwand $O(1)$ möglich sein, einen gegebenen Knoten aus der Liste zu entfernen. Dies ist mit drei (!) Feldern der Dimension n möglich. Position i entspricht stets Knoten i.

Beispiel $n = 7$

Knoten	1	2	3	4	5	6	7
Grad	2	0	2	3	4	4	2

deg/Knoten	0/1	1/2	2/3	3/4	4/5	5/6	6/7
degstart	2	-	1	4	5	-	-
vorg	0	0	1	0	0	5	3
nachf	3	0	7	0	6	0	0

Operationen:

(i) entferne einen Knoten, z.B. Knoten 3. Aufwand $\mathcal{O}(1)$, wenn Grad bekannt ist.

deg/Knoten	0/1	1/2	2/3	3/4	4/5	5/6	6/7
degstart	2	-	1	4	5	-	-
vorg	0	0	1	0	0	5	1
nachf	7	0	7	0	6	0	0

(ii) Ändern des Grades eines Knotens, z.B. Knoten 5 bekommt Grad 3. Füge in neue Liste ein, lösche aus alter Liste.

Knoten	1	2	3	4	5	6	7
Grad	2	0	2	3	3	4	2

Aufwand: $\mathcal{O}(1)$, wenn der alte Grad bekannt ist.

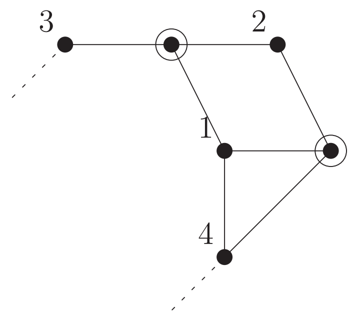
deg/Knoten	0/1	1/2	2/3	3/4	4/5	5/6	6/7
degstart	2	-	1	5	6	-	-
vorg	0	0	1	5	0	0	1
nachf	7	0	7	0	4	0	0

Folgerung: Es gelte $\mathit{mindeg} \leq c$, $|R| \leq d$ in allen Stufen des MD-Algorithmus. Verwendet man die obige Datenstruktur, so gilt für den Aufwand für (1) und (2):

(1) : $\mathcal{O}(c)$ (suche erste nichtleere Liste)
(2) : $\mathcal{O}(d)$

$\Rightarrow$ Gesamtaufwand $\mathcal{O}(n(c+d))$.

Zum Abschluss besprechen wir noch eine letzte Verbesserung für den MD-Algorithmus.
Ausgangsüberlegung: Zwei Knoten haben „die gleiche Nachbarschaft in $\mathcal{G}_k$“.



$$\mathit{Reach}(1,S) = \{2,3,4\} \quad \mathit{Reach}(2,S) = \{1,3,4\}.$$

Es gilt dann $\deg_{\mathcal{G}_k}(1) = \deg_{\mathcal{G}_k}(2)$, was sich auch in den nächsten Schritten nicht mehr ändert. Zusätzlich gilt:
Wenn ein Knoten derjenige mit minimalem Grad wird (z.B. $1 = \pi(k)$), so kann man $\pi(k+1) = 2$ nehmen (siehe Satz 3.3.27 unten). $\rightsquigarrow$ Einsparung beim Aufdatieren des Graphen und bei der Minimumsbestimmung. Wir gehen dies jetzt systematisch an.

3.3.23 Definition
Sei $G = (E, V)$, $S \subseteq V$, $T = V \setminus S$ und $x, y \in T$. Dann heißen x, y *nicht unterscheidbar bezüglich* S („*nubS*“), falls gilt

$$\mathit{Reach}(x,S) \cup \{x\} = \mathit{Reach}(y,S) \cup \{y\}.$$

3.3.24 Satz
Zu $G = (V, E)$ seien $x \neq y$ $nubS$ und $S \subseteq S' \subseteq V$, $x, y \notin S'$. Dann sind x, y $nubS'$.

Beweis: Es genügt zu zeigen, dass

$$Reach(x, S') \cup \{x\} \subseteq Reach(y, S') \cup \{y\}$$

gilt (wegen Symmetrie gilt dann auch „$\supset$"). Nach Voraussetzung gilt

$$Reach(x, S) \cup \{x\} = Reach(y, S) \cup \{y\}.$$

Klar ist: $y \in Reach(x, S')$, da $y \notin S'$. Sei nun $z \in Reach(x, S')$, $z \neq y$. Dann existiert ein Kantenzug $(x, s_1, \ldots, s_n, z)$, alle $s_i \in S'$.

1. Fall: Alle $s_i \in S$ oder $n = 0$. Dann ist $z \in Reach(x, S) \subseteq Reach(y, S) \cup \{y\} \Rightarrow z \in Reach(y, S) \Rightarrow z \in Reach(y, S')$.

2. Fall: Es existiert ein kleinstes i, so dass $s_i \in S' \setminus S$ ist. Dann ist $s_i \in Reach(x, S) \subseteq Reach(y, S) \cup \{y\}$ und damit $z \in Reach(y, S')$. Insgesamt ist also

$$Reach(x, S') \cup \{x\} \subseteq Reach(y, S') \cup \{y\}.$$

□

3.3.25 Lemma
Sei $y \notin S$, $S' = S \cup \{y\}$. Dann gilt

$$Reach(x, S') = \begin{cases} Reach(x, S) & \text{für } x \notin Reach(y, S) \\ Reach(x, S) \cup Reach(y, S) \setminus \{x, y\} & \text{sonst.} \end{cases}$$

Beweis: Trivial. □

3.3.26 Korollar
In der Situation von Lemma 3.3.25 gilt

$$|Reach(x, S')| \geq |Reach(x, S)| - 1.$$

Beweis: Im Fall $x \notin Reach(y, S)$ gilt $|Reach(x, S')| = |Reach(x, S)|$. Andernfalls folgt wegen $x \in Reach(y, S)$, $x \notin Reach(x, S)$ die Behauptung. □

3.3.27 Satz
Sei $x \neq y \notin S$, x, y $nubS$. Weiter besitze x minimalen Grad im entsprechenden Eliminationsgraphen, d.h.

$$|Reach(x,S)| \leq |Reach(z,S)| \quad \forall z \in T.$$

Sei $S' = S \cup \{x\}$, $T' = T \setminus \{x\}$. Dann besitzt y minimalen Grad im nächsten Eliminationsgraphen, d.h. es gilt

$$|Reach(y,S')| \leq |Reach(z,S')| \quad \forall z \in T'.$$

Beweis: Es gilt

$$Reach(y,S') = Reach(y,S) \setminus \{x\} \text{ (da } x, y \text{ } nubS)$$

Also gilt für $z \in T$

$$\begin{aligned} |Reach(y,S')| &= |Reach(y,S)| - 1 \\ &= |Reach(x,S)| - 1 \\ &\leq |Reach(z,S)| - 1 \\ &\leq |Reach(z,S')|. \end{aligned}$$

□

Folgerung: Wird $x = \pi(k)$ als Knoten mit minimalem Grad gefunden, kann man für die MD-Anordnung als nächstes alle Knoten y mit x, y $nubS_k$ wählen. Es entfällt also die Suche nach Knoten mit kleinstem Grad.

Wie findet man *nub*-Knoten einfach? Für die Praxis benötigt man nicht notwendig alle. Das folgende Kriterium hat sich bewährt.

3.3.28 Satz
Sei $S \subseteq V$, C_1, C_2 zwei Zusammenhangskomponenten von $G|_S$. Sei $R_1 = adj(C_1)$, $R_2 = adj(C_2)$. Dann sind alle Knoten y mit

$$y \in Y = \{y \in R_1 \cap R_2 : adj(y) \subseteq R_1 \cup R_2 \cup C_1 \cup C_2\}$$

$nubS$ und $Reach(y,S) \cup \{y\} = R_1 \cup R_2$.

Beweis:

„$\subseteq$“: $y \in R_1 \cap R_2 \Rightarrow y \in R_1 \cup R_2$. Sei $z \in Reach(y,S)$ und $(y, s_1, \ldots, s_m, z)$ der zugehörige Kantenzug. Falls $m = 0$, ist

$$z \in adj(y) \setminus S \subseteq R_1 \cup R_2 \cup C_1 \cup C_2 \setminus S \subseteq R_1 \cup R_2.$$

Falls $m > 0$, ist $s_1 \in adj(y) \cap S \subseteq C_1 \cup C_2$. Also ist $\{s_1, s_2, \ldots, s_m\} \subseteq C_1$ oder $\subseteq C_2$. Also $z \in R_1 \cup R_2$.

„$\supseteq$“: $z \neq y \in R_1 \cup R_2 \Rightarrow$ o.B.d.A. $z \in R_1$. Da C_1 in $G|_S$ zusammenhängend ist, ist y mit z über C_1 verbunden, also auch über S und damit $z \in \mathit{Reach}(y, S)$.

$\square$

Damit erhalten wir folgende Form des MD-Algorithmus

3.3.29 Algorithmus
(Berechnet die MD-Anordnung)

$S = \emptyset, T = \{1, \ldots, n\}$
while $T \neq \emptyset$ **do**
  finde Knoten p aus T mit minimalem Grad
  nummeriere p und alle bekannten, von p *nubS* Knoten als nächste für die MD-Anordnung
  lege alle in S; entferne alle aus T
  bilde neue Superknoten, datiere $\mathcal{G}$ auf (wie Algorithmus 3.3.21)
  identifiziere dabei Knoten *nubS* nach Satz 3.3.28
end while

Finale Bemerkung: Knoten *nubS* verwaltet man ebenfalls als Superknoten, d.h. $\mathcal{G}$ ist Quotientengraph bezüglich der Zusammenhangskomponenten in S *und* bezüglich der Mengen von *nubS* Knoten.

Abschnitt 3.4

One-way-Dissection und Nested Dissection

3.4.1 Definition
$A \in \mathbb{R}^{n\times n}$ besitzt *Blockdiagonalform mit Rand*, falls

$$(3.4.1) \qquad A = \begin{bmatrix} A_{11} & & & & A_{1N} \\ & A_{22} & & 0 & \vdots \\ & & \ddots & & \vdots \\ & 0 & & A_{N-1,N-1} & A_{N-1,N} \\ A_{N1} & \dots & \dots & A_{N,N-1} & A_{NN} \end{bmatrix}$$

mit $A_{ii} \in \mathbb{R}^{n_i\times n_i}$, $\sum_{i=1}^{N} n_i = n$ gilt.

Ist A symmetrisch, so gilt $A_{N_i} = A_{N_i}^T$, $A_{ii} = A_{ii}^T$. Die Form (3.4.1) ist günstig für (Block-) Gauß-Elimination bzw. Cholesky-Faktorisierung wegen

3.4.2 Satz
Falls alle nachstehenden Matrizen $D_{ii} \in \mathbb{R}^{n_i\times n_i}$, $i = 1,\dots,N-1$ regulär sind, gilt

$$A = \begin{bmatrix} I & & & \\ & I & & \\ & & \ddots & \\ A_{N1}D_{11}^{-1} & \dots & A_{N,N-1}D_{N-1,N-1}^{-1} & I \end{bmatrix} D \begin{bmatrix} I & & D_{11}^{-1}A_{1N} \\ & \ddots & \vdots \\ & & \ddots \; D_{N-1,N-1}^{-1}A_{N-1,N} \\ & & I \end{bmatrix}$$

mit $D_{ii} = A_{ii}$, $i = 1,\dots,N-1$, $D_{NN} = A_{NN} - \sum_{j=1}^{N-1} A_{Nj}D_{jj}^{-1}A_{jN}$.

Beweis: Nachrechnen! □

Bemerkung: Ist A spd, so ist D_{ii} spd, $i = 1,\dots,N$. Insbesondere sind alle D_{ii} dann regulär. S. Übung.

Die Darstellung aus Satz 3.4.2 ist eine Block-Faktorisierung. Zu berechnen ist lediglich D_{NN}:

for $j = 1$ to $N-1$ **do**
  löse die n_N Systeme $D_{jj}V_j = A_{jN}$ $\quad \{D_{jj}$ ist vermutlich dünn besetzt$\}$

end for
setze $D_{NN} = A_{NN} - \sum_{j=1}^{N-1} A_{Nj}V_j$ $\{A_{Nj}$ ist vermutlich dünn besetzt$\}$

Die Lösung von $Ax = b$ erfolgt dann so:

$$(L(D(Ux)) = b) \text{ (Bezeichnung } x_i \mathrel{\widehat{=}} \text{ Blockkomponente von } x).$$

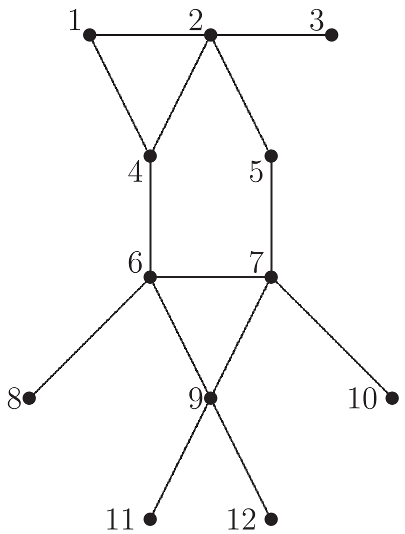
3.4.3 Algorithmus
(Lösen von $Ax = b$ bei Block-Faktorisierung)

$z = b$
for $j = 1$ to $N - 1$ **do** $\{\widehat{=} L\}$
  löse $D_{jj}w_j = z_j$
  $z_N = z_N - A_{Nj}w_j$
end for
for $j = 1$ to N **do** $\{\widehat{=} D\}$
  löse $D_{jj}y_j = z_j$
end for
for $j = 1$ to $N - 1$ **do** $\{\widehat{=} U\}$
  löse $D_{jj}v_j = A_{jN}y_N$
  $x_j = y_j - v_j$
end for
$x_N = y_N$

Weil die D_{jj} im Allgemeinen wieder dünn besetzt sind, wird man die Systeme mit D_{jj} über eine Faktorisierung von D_{jj} als db Matrizen lösen.

3.4.4 Definition
Sei $G = (V, E)$ ein zusammenhängender Graph und $S \subseteq V$. Dann heißt S *Separator* von G, falls $G|_{V \setminus S}$ nicht zusammenhängend ist.

Beispiel:



$\{9\}$ ist Separator ($\rightsquigarrow$ 3 Zusammenhangskomponenten)
$\{2\}$ ist Separator ($\rightsquigarrow$ 2 Zusammenhangskomponenten)
$\{6,7\}$ ist Separator ($\rightsquigarrow$ 4 Zusammenhangskomponenten)
$\{2,6,7\}$ ist Separator ($\rightsquigarrow$ 6 Zusammenhangskomponenten)

3.4.5 Lemma
A sei symmetrisch, $G(A)$ sei zusammenhängend, $S \subseteq V$ sei Separator, die Mengen $V_1, V_2, \ldots, V_m$ seien (Knotenmengen der) Zusammenhangskomponenten von $G|_{V \setminus S}$. π sei eine Permutation, die zuerst alle Knoten von V_1, dann alle in V_2, …, dann alle in V_m und dann alle von S anordnet. P sei zugehörige Permutationsmatrix. Dann besitzt PAP^T Blockdiagonalform mit Rand

$$PAP^T = \begin{bmatrix} A_{11} & & & A_{1,m+1} \\ & \ddots & & \vdots \\ & & A_{mm} & \vdots \\ A_{m+1,1} & \cdots & \cdots & A_{m+1,m+1} \end{bmatrix}$$

mit $A_{ii} \in \mathbb{R}^{n_i \times n_i}$, $n_i = |V_i|$, $i = 1, \ldots, m$, $n_{m+1} = |S|$.

Beweis: Trivial, da ein Knoten aus V_i mit keinem aus V_j, $j \neq i$ adjazent ist.
$\square$

Wünschenswert sind Separatoren S mit $|S|$ klein und m groß.

Beispiel von vorhin: $S = \{2, 6, 7\}$
$V_1 = \{1, 4\}, V_2 = \{3\}, V_3 = \{5\}, V_4 = \{8\}, V_5 = \{9, 11, 12\}, V_6 = \{10\}$.

	1	4	3	5	8	11	12	9	10	2	6	7
1	x	x								x		
4	x	x								x	x	
3			x							x		
5				x						x		x
8					x						x	
11						x		x				
12							x	x				
9						x	x	x			x	x
10									x			x
2	x	x	x	x						x		
6		x			x			x			x	x
7				x					x		x	x

Levelmengen liefern „gute“ Separatoren.

3.4.6 Lemma
$S_1, \ldots, S_k$ seien die Levelmengen eines Graphen $G = (V, E)$ startend mit S_1. Dann gilt

(i) S_i ist Separator für $i = 2, \ldots, k-1$.

(ii) Für jede Indexmenge $I \subseteq \{2, \ldots, k-1\}$ mit der Eigenschaft $i, j \in I, i \neq j \Rightarrow |i - j| > 1$ ist
$$S = \bigcup_{i \in I} S_i$$
Separator, so dass $G|_{V \setminus S}$ mindestens $|I| + 1$ Zusammenhangskomponenten besitzt.

Beweis:

(i) ist Spezialfall von (ii).

(ii) Sei $|I| = t \geq 1$ und $I = \{i_1, \ldots, i_t\}$ mit $i_1 < i_2 < \ldots < i_t$. Dann ist klar:

$C_1 = S_1 \cup \ldots \cup S_{i_1 - 1}$ besteht aus so vielen Zusammenhangs-Komponenten in $G|_{V \setminus S}$ wie $G|_{S_1}$,

$C_2 = S_{i_1 + 1} \cup \ldots \cup S_{i_2 - 1}$ besteht aus so vielen Zusammenhangs-Komponenten wie $G|_{S_{i_1+1}}$,

$\vdots$

$C_{t+1} = S_{i_t+1} \cup \ldots \cup S_k$ besteht aus so vielen Zusammenhangs-Komponenten wie $G|_{S_{i_t+1}}$.

Außerdem sind die C_i untereinander nicht verbunden.

□

3.4.7 Algorithmus
(One-Way-Dissection, George 1980)

{Eingabe ist ein zusammenhängender Graph G, Ausgabe ist eine Anordnung als Permutation π auf der Knotenmenge}

bestimme pseudoperipheren Knoten v
bestimme alle Levelmengen $S_1, \ldots, S_k$ mit $S_1 = \{v\}$
nehme *geeignetes* $I \subseteq \{1, \ldots, k\}$ und setze $S = \bigcup_{i \in I} S_i$
finde die Permutation π wie folgt:

```
p = 0
for i = 1 to k do
  if i ∉ I then
    for all Knoten w ∈ S_i do
      p++
      π(w) = p
    end for
  end if
end for
for i = 2 to k − 1 do
  if i ∈ I then
    for all Knoten w ∈ S_i do
      p++
      π(w) = p
    end for
  end if
end for
```

Wie findet man geeignete Menge I?

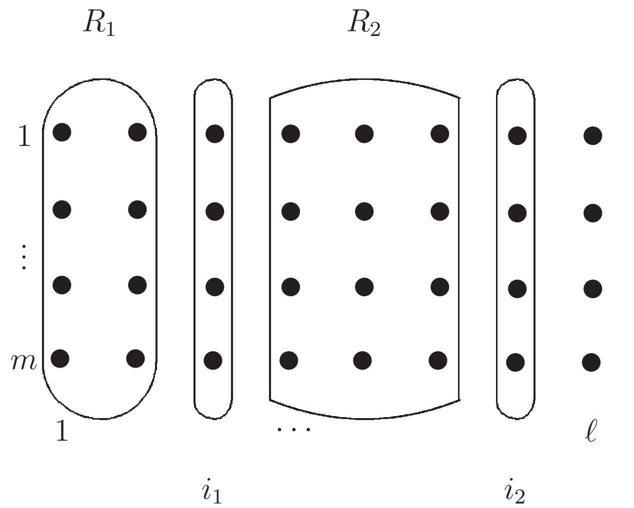
Empfehlung (empirisch): setze $\delta = \sqrt{\frac{3\frac{n}{k}+13}{2}}$ und nehme:

$$i \in I \Longleftrightarrow i \approx \mu\delta, \ \mu \in \mathbb{N}.$$

Genauer:

$i \in I \Longleftrightarrow i = \lfloor \mu\delta + 0.5 \rfloor, \ \mu = 1, \ldots, m \ (\lfloor m\delta + 0.5 \rfloor < k \leq \lfloor (m+1)\delta + 0.5 \rfloor)$.

3.4.8 Beispiel
G sei ein $m \times \ell$-Gitter



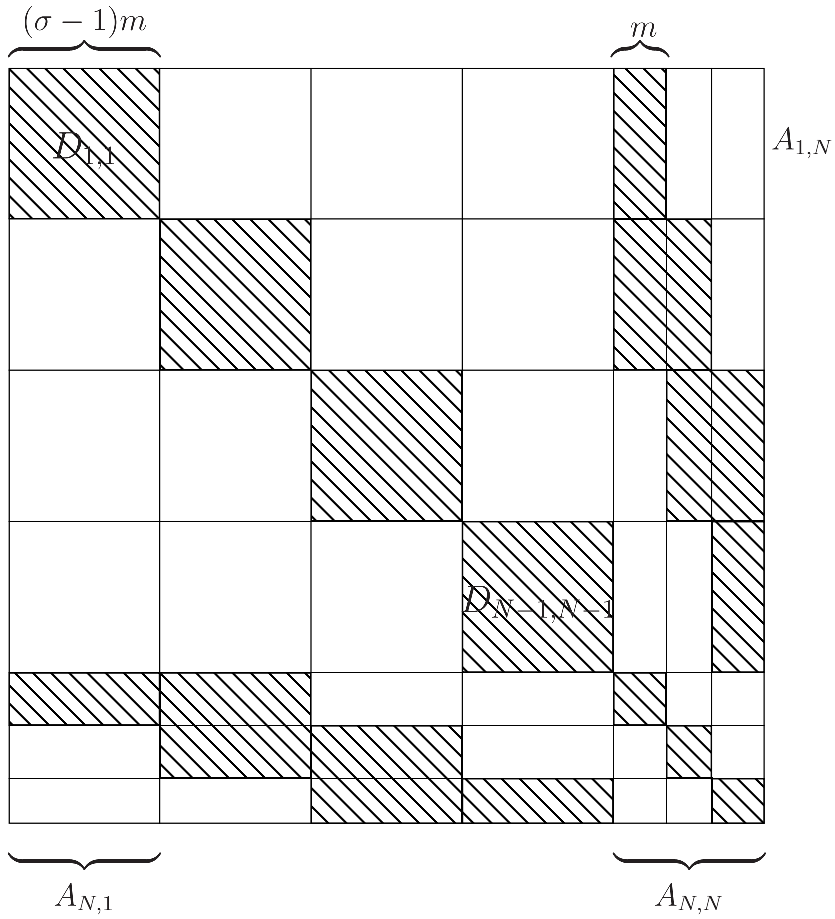
Im Gegensatz zu Algortihmus 3.4.7 nehmen wir $S_1 =$ erste Spalte des Gitters $\rightsquigarrow S_k = k$-te Spalte des Gitters.

Ansatz:

$$i \in I \Longleftrightarrow i = \sigma j, \ \sigma \geq 2, \ j = 1, \ldots, \frac{\ell}{\sigma} - 1.$$

(Wir setzen voraus: σ teilt ℓ, $N := \frac{\ell}{\sigma}$).

Entstehende Blockdiagonalgestalt mit Rand



$$N-1=\frac{\ell}{\sigma}$$

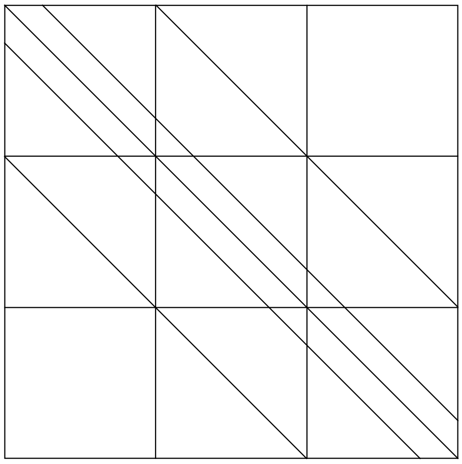
Beachte: A_{Nj}, A_{jN}, A_{NN} sind selbst wieder strukturiert.

A_{jN}: nur ≤ 2 besetzte Blockspalten der Breite m,

A_{Nj}: nur ≤ 2 besetzte Blockzeilen der Höhe m,

A_{NN}: blockdiagonal mit $(\frac{\ell}{\sigma}-1)$ Blöcken der Größe m.

Jede Nichtnullspalte von A_{jN} (bzw. -zeile von A_{Nj}) besitzt genau ein Nichtnullelement.
Wir nummerieren außerdem innerhalb jeder Zusammenhangskomponente R_i die Knoten zeilenweise. $\rightsquigarrow$ jedes D_{ii} ist Bandmatrix mit halber Bandbreite $\sigma-1$.

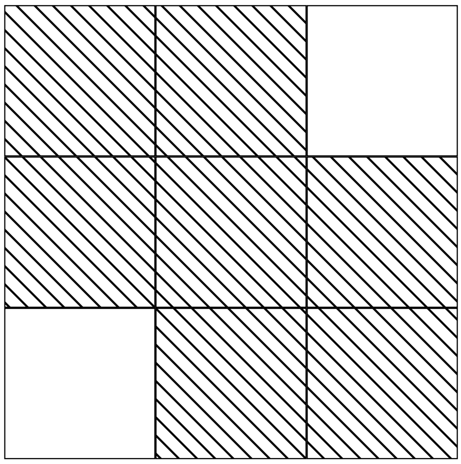


D_i

Weitere Beobachtung:

$$D_{NN} = A_{NN} - \sum_{j=1}^{N-1} A_{Nj} D_{jj}^{-1} A_{jN}$$

besitzt Blocktridiagonalgestalt; insbesondere die halbe Bandbreite $2m$.



Wir analysieren jetzt die Größenordnung aller Berechnungen in Algorithmus 3.4.3, inklusive der Berechnung von D_{NN}. Dazu

3.4.9 Lemma
$C \in \mathbb{R}^{k\times k}$ besitze halbe Bandbreite b. Dann besitzt L in $C = LDL^T$ die halbe Bandbreite b. Die Berechnung von L, D kostet $\mathcal{O}(kb^2)$; jede Lösung von $LDL^Tx = c$ kostet $\mathcal{O}(kb)$.

Beweis: Bekannt. □

Berechnung von D_{NN}:

1. Faktorisierung von D_{jj}, $j = 1, \ldots, N-1$
   pro j: $\mathcal{O}((\sigma-1)m(\sigma-1)^2)$.
2. Lösen von $D_{jj}V_j = A_{jN}$ (maximal $2m$ Stück)
   pro j: $\mathcal{O}((2m(\sigma-1)m(\sigma-1))$, $j = 1, \ldots, N-1$
3. Multiplikation $A_{Nj}V_j$ (nur Nichtnullblöcke)
   Jede Zeile von A_{Nj} hat höchstens eine Nichtnull, $j = 1, \ldots, N-1$
   pro j: $\mathcal{O}(m^2)$

Gesamtaufwand:

$$\begin{aligned} & \mathcal{O}(m(\sigma-1)^3)(N-1) + \mathcal{O}(m^2(\sigma-1)^2)(N-1) + \mathcal{O}(m^2)(N-1) \\ = \; & \mathcal{O}\left(m\frac{\sigma^3\ell}{\sigma}\right) + \mathcal{O}\left(m^2\frac{\sigma^2\ell}{\sigma}\right) + \mathcal{O}\left(m^2\frac{\ell}{\sigma}\right) \quad (\text{denn } N-1 = \frac{\ell}{\sigma}) \\ = \; & \mathcal{O}(m\sigma^2\ell) + \mathcal{O}(m^2\sigma\ell) \end{aligned}$$

Durchführung von Algorithmus 1:
1. Schleife ($\widehat{=}\, L$): $\mathcal{O}(m\sigma^2)$, $j = 1, \ldots, N-1$
2. Schleife ($\widehat{=}\, D$): $\mathcal{O}(m\sigma^2)$, $j = 1, \ldots, N-1$
$\mathcal{O}((\frac{\ell}{\sigma}-1)mm^2)$ (für D_{NN} inklusive Faktorisierung (als Bandmatrix))
3. Schleife ($\widehat{=}\, U$): $\mathcal{O}(m\sigma^2)$, $j = 1, \ldots, N-1$

Insgesamt für Algorithmus 1 also

$$\mathcal{O}\left(m\sigma^2\frac{\ell}{\sigma}\right) + \mathcal{O}\left(m\sigma^2\frac{\ell}{\sigma}\right) + \mathcal{O}\left(\frac{\ell}{\sigma}m^3\right) + \mathcal{O}\left(m\sigma^2\frac{\ell}{\sigma}\right) = \mathcal{O}(m\sigma\ell) + \mathcal{O}\left(\frac{\ell}{\sigma}m^3\right)$$

Mit der Berücksichtigung der Berechnung von D_{NN} beträgt der Aufwand also

$$\begin{aligned} & \mathcal{O}(m\sigma^2\ell) + \mathcal{O}(m^2\sigma\ell) + \mathcal{O}(m\sigma\ell) + \mathcal{O}\left(\frac{\ell}{\sigma}m^3\right) \\ = \; & \mathcal{O}(m\sigma^2\ell) + \mathcal{O}(m^2\sigma\ell) + \mathcal{O}\left(\frac{\ell}{\sigma}m^3\right) \end{aligned}$$

Wähle nun $\sigma = \sigma(m, \ell)$ so, dass der dominierende $\mathcal{O}$-Term möglichst klein wird.

Spezialfall: $m = \ell$
Ansatz: $\sigma = \ell^\alpha$

$$\begin{aligned}\mathcal{O}(m\sigma^2\ell) &= \mathcal{O}(\ell^{2+2\alpha}) \\ \mathcal{O}(m^2\sigma\ell) &= \mathcal{O}(\ell^{3+\alpha}) \\ \mathcal{O}\left(\frac{\ell}{\sigma}m^3\right) &= \mathcal{O}(\ell^{4-\alpha})\end{aligned}$$

$\max\{2+2\alpha, 3+\alpha, 4-\alpha\}$ wird minimiert für $\alpha = \frac{1}{2}$.

Gesamtaufwand dann: $\mathcal{O}(\ell^{7/2})$.

Zum Vergleich: Faktorisierung bei zeilenweiser Anordnung des gesamten Gitters ($\rightsquigarrow$ halbe Bandbreite ist ℓ) ergibt Aufwand $\mathcal{O}(\ell^2\ell^2) = \mathcal{O}(\ell^4)$.
Praktische Vorteile von One-Way-Dissection gegenüber MD:

a) einfacher zu implementieren,

b) σ kann auch bezüglich des Speicheraufwandes optimiert werden. $\rightsquigarrow$ günstig bei knappem Speicher.

Nested Dissection: Rekursive Anwendung von One-Way-Dissection.
Idee: Finde „kleinen" Separator S, so dass $G|_{V\setminus S}$ (möglichst) zwei, ungefähr gleich große Zusammenhangskomponenten R_1, R_2 besitzt. Wiederhole dasselbe für $G|_{R_1}$ und $G|_{R_2}$.

Entstehende Struktur von PAP^T:

$$PAP^T = \begin{bmatrix} A_{11} & 0 & A_{13} \\ 0 & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix}$$

wobei A_{11}, A_{22} dieselbe Gestalt wie PAP^T besitzen. Ab jetzt gehen wir davon aus, dass immer genau 2 Zusammenhangskomponenten R_1, R_2 entstehen.

3.4.10 Algorithmus
(Nested-Dissection, George/Liu 1978)

{ Eingabe ist ein zusammenhängender Graph G, Ausgabe ist eine Anordnung als Permutation π auf der Knotenmenge}

bestimme pseudoperipheren Knoten v
bestimme alle zugehörigen Levelmengen $S_1, \ldots, S_\ell$
if $\ell \leq 2$ **then**
  $\pi = id$
else
  setze $j = \lfloor(\ell + 1)/2\rfloor$, setze $S = \{y \in S_j : adj(y) \cap S_{j+1} \neq \emptyset\}$
  {S ist Separator}
  setze $R_1 = S_1 \cup \ldots \cup S_{j-1} \cup S_j \setminus S$, $R_2 = S_{j+1} \cup \ldots \cup S_\ell$
  $\pi 1$ = Ergebnis von Algorithmus 3.4.10 bei Eingabe $G|_{R_1}$
  $\pi 2$ = Ergebnis von Algorithmus 3.4.10 bei Eingabe $G|_{R_2}$
  $\pi = (\pi 1, \pi 2, id|_S)$ {zuerst $\pi 1$, dann $\pi 2$, dann Identität auf Separator}
end if

Nested Dissection ist bei planaren Graphen häufig „optimal“ im $\mathcal{O}$-Sinne. Wir weisen dies nach für das $\ell \times \ell$-Gitter aus Beispiel 3.4.8.

3.4.11 Lemma
$k \in \mathbb{R}^+$ sei fest, $n \in \mathbb{N}$.

(i) Es gelte $f(n) = f(\frac{n}{2}) + kn^3 + \mathcal{O}(n^2 \log n)$. Dann gilt

$$f(n) \leq \frac{8}{7}kn^3 + \mathcal{O}(n^2 \log n).$$

(ii) Es gelte $f(n) = 2f(\frac{n}{2}) + kn^3 + \mathcal{O}(n^2 \log n)$. Dann gilt

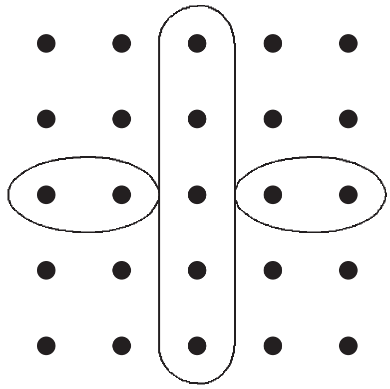
$$f(n) \leq \frac{4}{3}kn^3 + \mathcal{O}(n^2 \log n).$$

(iii) Es gelte $f(n) = 4f(\frac{n}{2}) + kn^3 + \mathcal{O}(n^2)$. Dann gilt

$$f(n) \leq 2kn^3 + \mathcal{O}(n^2 \log n).$$

Beweis: Übungsaufgabe □

Das Gitter sei diesmal quadratisch:

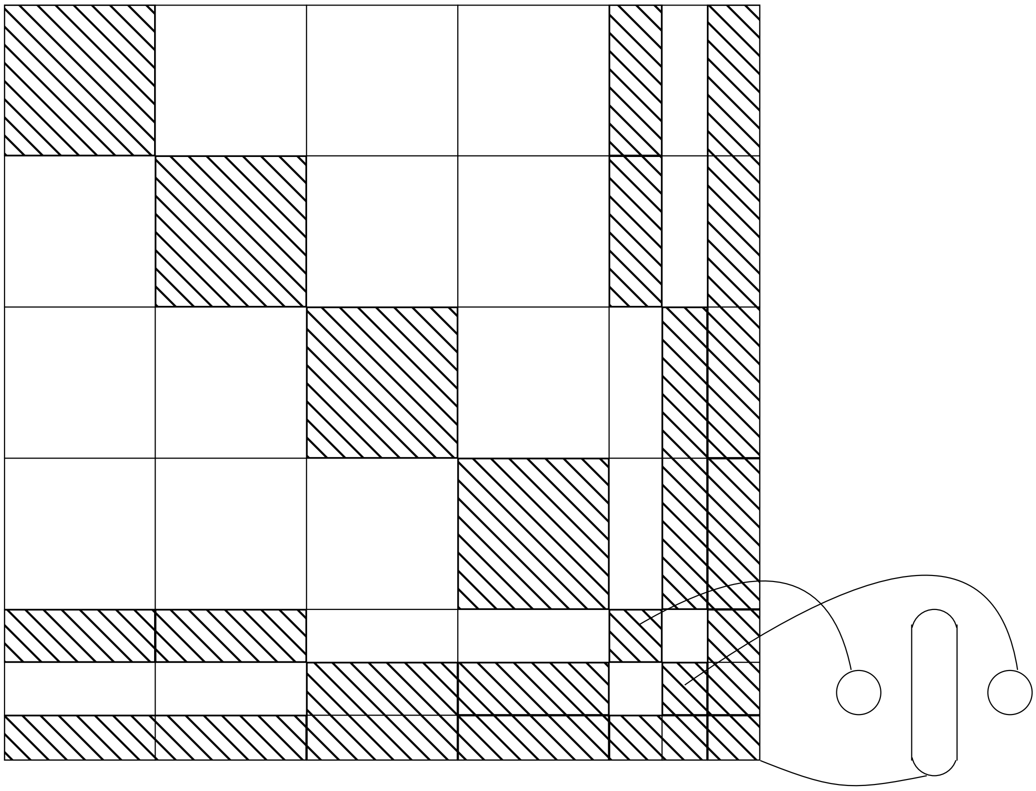


Wir betrachten stets zwei Rekursionsschritte von Algorithmus 3.4.10 zusammen; Separatoren wie in der Skizze (also nicht Levelmengen zu pseudoperipheren Punkten wie in Algortihmus 3.4.10).
In der Aufwandsanalyse betrachten wir für jede Zusammenhangskomponente den Aufwand für die Faktorisierung des Diagonalblocks in der Matrix ($\widehat{=} A_{11}, A_{22}$) zusammen mit der Transformation der zugehörigen Randblöcke (Anteil $A_{31}A_{11}^{-1}A_{13}$ bzw. $A_{32}A_{22}^{-1}A_{23}$.)

Randblock $\widehat{=}$ Kopplungen zu Separatoren. Bei mehrfacher Rekursion entstehen Kopplungen an verschieden vielen Rändern:

Typ 2: Kopplung an 2 Rändern
Typ 3: Kopplung an 3 Rändern
Typ 4: Kopplung an 4 Rändern

Matrix nach 2 Rekursionsschriten:



Für jeden Typ bezeichnen wir den Aufwand mit $\theta(n, i)$ (n Gitterbreite; i Typ)

$i = 0 \mathrel{\widehat{=}}$ Anfangsgitter
$i = 1$ kommt nicht vor.

Damit erhalten wir folgende Rekursionen im Fall $n = 2^N - 1$

$$
\left\{
\begin{aligned}
\theta(n, 0) &= 4\theta(\tfrac{n-1}{2}, 2) + \Phi_0(n) \\
\theta(n, 2) &= \theta(\tfrac{n-1}{2}, 4) + 2\theta(\tfrac{n-1}{2}, 3) + \theta(\tfrac{n-1}{2}, 2) + \Phi_2(n) \\
\theta(n, 3) &= 2\theta(\tfrac{n-1}{2}, 4) + 2\theta(\tfrac{n-1}{2}, 3) + \Phi_3(n) \\
\theta(n, 4) &= 4\theta(\tfrac{n-1}{2}, 4) + \Phi_4(n)
\end{aligned}
\right. \tag{3.4.2}
$$

$\Phi_0, \Phi_2, \Phi_3, \Phi_4$: Aufwand für Faktorisierung des „Separatorblocks“ ($\mathrel{\widehat{=}} A_{33}$) ist jeweils $\leq kn^3$.

3.4.12 Satz
Nested Dissection für Beispiel 3.4.8 (mit $m = \ell = n$) erfordert Aufwand $\mathcal{O}(n^3)$.

Beweis: Zur Vereinfachung sind wir etwas schlampig: Wir verwenden (3.4.2) so, als ob rechts statt $\frac{n-1}{2}$ immer $\frac{n}{2}$ stünde. Dann können wir Lemma 3.4.11 anwenden:
Aus (3.4.2) und Lemma 3.4.11 (iii):

$$\theta(n,4) = 2kn^3 + \mathcal{O}(n^2 \log n).$$

Aus (3.4.2) und Lemma 3.4.11 (ii):

$$\theta(n,3) = 2kn^3 + \mathcal{O}(n^2 \log n).$$

Aus (3.4.2) und Lemma 3.4.11 (i):

$$\theta(n,2) = \frac{7}{4}kn^3 + \mathcal{O}(n^2 \log n).$$

Und damit

$$\theta(n,0) = \frac{7}{8}kn^3 + kn^3 = \mathcal{O}(n^3).$$

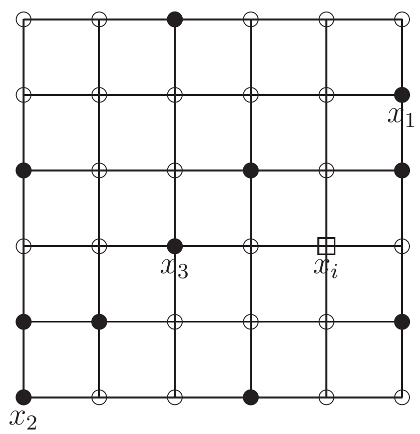
□

Höchstinteressanterweise ist dieser Aufwand asymptotisch optimal.

3.4.13 Lemma
Sei $G = (V, E)$ der Graph des $n \times n$-Gitters. Es seien weiter $x_1, \ldots, x_m$, $(m = n^2)$ die gewählten Pivotknoten. Dann existiert ein $i \in \{1, \ldots, m-n\}$ mit

$$|Reach(x_i, \{x_1, \ldots, x_{i-1}\})| \geq n - 1.$$

Beweis:



Sei i die kleinste Nummer, so dass mit i erstmalig eine ganze Zeile oder Spalte des Gitters in $\{x_1, \dots, x_i\}$ liegt. O.B.d.A sei dies eine Spalte. Damit sind mindestens $n-1$ Zeilen nicht ganz in $\{x_1, \dots, x_i\}$. In jeder dieser Zeilen ist wenigstens ein Knoten $\notin \{x_1, \dots, x_i\}$ und mit x_i über $\{x_1, \dots, x_{i-1}\}$ verbunden. $\square$

3.4.14 Satz
Bei jeder Anordnung der Knoten besitzt die Gauß-Elimination zur Faktorisierung der Matrix $A = A^T$ mit $G(A) = n \times n$-Gitter den Aufwand $\Omega(n^3)$.

Beweis: Nach Lemma 3.4.13 existiert ein i, so dass nach dem i-ten Eliminationsschritt $A^{(i)}(i:m, i:m)$ eine vollbesetzte $(n-1) \times (n-1)$-Teilmatrix besitzt (Clique). Für die Faktorisierung dieses Teils ist der Aufwand bereits $\Omega(n^3)$. $\square$

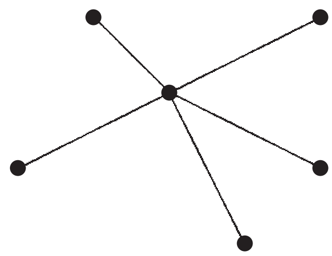
Abschnitt 3.5

Quotientenbäume

Ausgangssituation: $\mathcal{V}$ sei Partition von V in $G = (V, E)$, so dass $\mathcal{G} = G/\mathcal{V}$ ein Baum ist.
Erinnerung an Satz 3.3.5: Bei MD (oder RCM)-Anordnung entsteht im Baum keinerlei fill-in.

3.5.1 Beispiel

a) $G(A)$ mit A blockdiagonal mit Rand, $\mathcal{V} = \{V_1, \ldots, V_N\}$ einzelne Blöcke. $\Rightarrow \mathcal{G}$ ist Stern (= spezieller Baum).

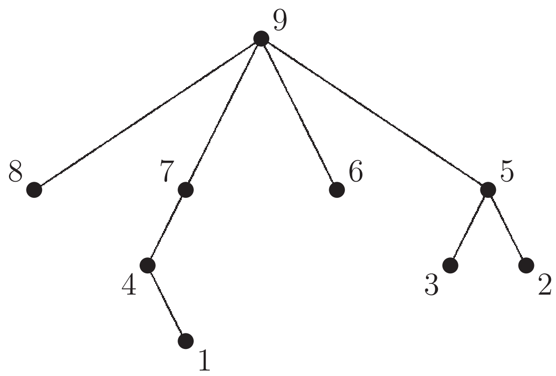


b) G ist angeordnet nach Levelmengen, $\mathcal{V} = \{S_1, \ldots, S_k\} \Rightarrow \mathcal{G}$ ist Kette (= spezieller Baum).

Es sei nun $\mathcal{G}$ ein Baum, und die Knoten in $\mathcal{V}$ seien entsprechend der MD-Anordnung nummeriert, $|\mathcal{V}| = m$. Dann heißt

- der Knoten m die *Wurzel* des Baumes,
- für $x \neq m$ der Knoten $y > x$ mit $\{x, y\} \in E$ *Vater* von x. Bezeichnung: $y = v(x)$.

Beispiel:



$v(1) = 4),\ v(2) = 5$, Wurzel ist 9.

In Übertragung von Satz 3.3.5 und Verallgemeinerung von Satz 3.4.2 gilt

3.5.2 Satz
$A \in \mathbb{R}^{n\times n}$ sei spd, $\mathcal{G} = G(A)/\mathcal{V}$ sei Baum, wobei Knoten von $\mathcal{V}$ nach der MD-Anordnung nummeriert seien. Dann existiert folgende Blockfaktorisierung von A:

$$A = \begin{bmatrix} A_{11} & \dots & A_{1N} \\ \vdots & & \vdots \\ A_{N1} & \dots & A_{NN} \end{bmatrix}, \qquad A_{ij} \in \mathbb{R}^{n_i \times n_j}, n_i = |V_i|$$

mit

$$A = \begin{bmatrix} I & & & 0 \\ A_{21}D_{11}^{-1} & \ddots & & \\ \dots & \ddots & \ddots & \\ A_{N1}D_{11}^{-1} & A_{N2}D_{22}^{-1} & \dots & I \end{bmatrix} D \begin{bmatrix} I & D_{11}^{-1}A_{21} & \dots & D_{11}^{-1}A_{N1} \\ & \ddots & \ddots & D_{22}^{-1}A_{N2} \\ & & \ddots & \vdots \\ & & & I \end{bmatrix} \tag{3.5.1}$$

wobei $A_{ij} \neq 0 \Longleftrightarrow i = v(j)$ oder $j = v(i)$ und

$$D_{ii} = A_{ii} - \sum_{j=1; v(j)=i}^{i-1} A_{ij}D_{jj}^{-1}A_{ji}. \tag{3.5.2}$$

Beweis: Wir rechnen blockweise nach:
1. Fall: $i > j$; rechte Seite von (3.5.1):

$$\sum_{k=1}^{j-1} A_{ik}D_{kk}^{-1}A_{kj} + A_{ij}.$$

Ein Term $A_{ik}D_{kk}^{-1}A_{kj}$ ist $\neq 0$, falls $i = v(k)$ und $j = v(k)$. Da m Wurzel des Baumes ist, existieren Kantenzüge von i nach m und von m nach j. Durch Hinzunahme von $\{i, k\}$ und $\{k, j\}$ entsteht ein Zyklus. Ein Baum ist aber zyklusfrei. Also sind $\{i, k\}$ und $\{k, j\}$ nicht beide Kanten. Damit gilt $A_{ik}D_{kk}^{-1}A_{kj} = 0$.
2. Fall: $i = j$; rechte Seite von (3.5.1):

$$\sum_{k=1}^{i-1} A_{ik}D_{kk}^{-1}A_{ki} + D_{ii} = A_{ii},$$

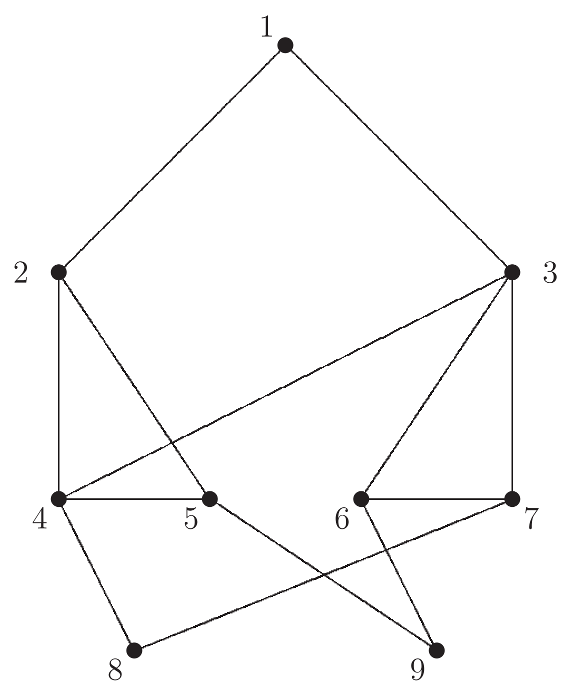
d.h. (3.5.2).
3. Fall: $i < j$ klar aus Symmetrie. $\square$

Die Lösung des LGS geht analog zu Algorithmus 3.4.3.

Erwünscht: Partition $\mathcal{V}$ mit möglichst vielen Teilmengen V_i, so dass $G/\mathcal{V}$ ein Baum ist.
Nach Beispiel 3.5.1b erhalten wir einen „groben“ Baum über die Levelmengen. Dann kann man aber noch „verfeinern“ nach folgender

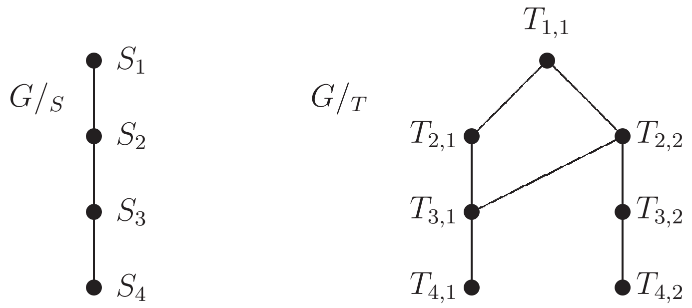
Idee: $G|_{S_i}$ besitze die Zusammenhangskomponenten T_{ij}, $j = 1, \ldots, s_i$. Der Quotientengraph bezüglich der T_{ij} ist nicht notwendig ein Baum. Er wird es aber, wenn man T_{ij} geeignet zusammenfasst.

3.5.3 Beispiel

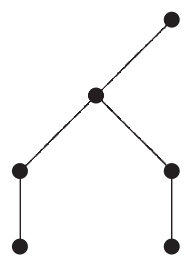


Levelmengen:

$$\begin{array}{lll}
S_1 = \{1\} & T_{1,1} = \{1\} & \\
S_2 = \{2,3\} & T_{2,1} = \{2\}, & T_{2,2} = \{3\} \\
S_3 = \{4,5,6,7\} & T_{3,1} = \{4,5\}, & T_{3,2} = \{6,7\} \\
S_4 = \{8,9\} & T_{4,1} = \{8\}, & T_{4,2} = \{9\}.
\end{array}$$



Beobachtung: Fasst man $T_{2,1}$ und $T_{2,2}$ zusammen, so entsteht als Quotientengraph ein Baum.



Ziel ist also, einige $T_{i,j}$ bei festem i so zu Mengen $V_{i,\nu}$ zusammenzufassen, dass ein Baum entsteht. Ein Baum entsteht, wenn ein $V_{i,\nu}$ zu genau einem $V_{i-1,\mu}$ adjazent ist.

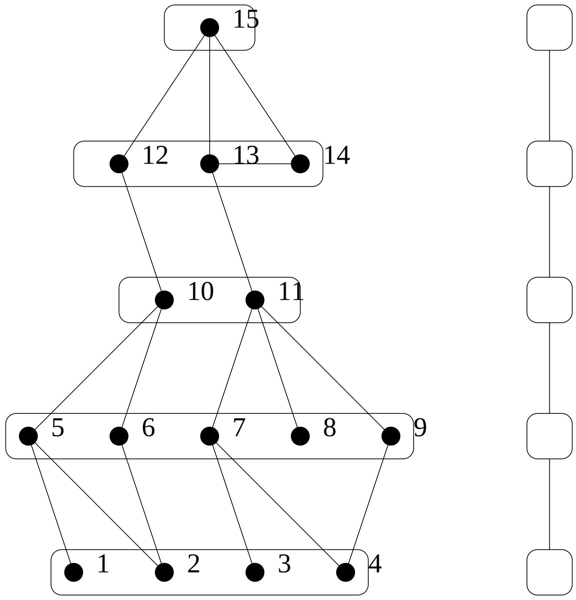
3.5.4 Algorithmus
(verfeinerte Quotientenbäume, George/Liu 1978)

{Gegeben: Partition $S_1, \dots, S_k$ bzgl. welcher der Quotientengraph ein Baum ist. (z.B. Levelmengen). Väter haben größere Nummern als Söhne (umgekehrt wie in Beispiel 3.5.3.).
Gesucht: für $i = 1, \dots, k$ Partitionierung $V_{i1}, \dots, V_{i,r_i}$ von S_i, so dass der Quotientengraph bezüglich aller $V_{i\nu}$ immer noch Baum ist.}

for $i = 1$ to k **do**
  bestimme die Zusammenhangskomponenten T_{ij}, $j = 1, \dots, s_i$ von G/S_i.
end for
setze $V_{1,j} = T_{1,j}$, $j = 1, \dots, s_1$
for $i = 2$ to k **do** {fasse T_{ij} geeignet zusammen}
  setze $\mathcal{P} = \{T_{ij}, j = 1, \dots, s_i\}$ { Partition von S_i}
  for $j = 1$ to r_{i-1} **do**
    entnehme alle $P \in \mathcal{P}$ mit $P \in adj(V_{i-1,j})$
    füge ihre Vereinigung wieder in $\mathcal{P}$ ein
  end for
  $\{V_{i1}, \dots, V_{i,r_i}\} = \mathcal{P}$
end for

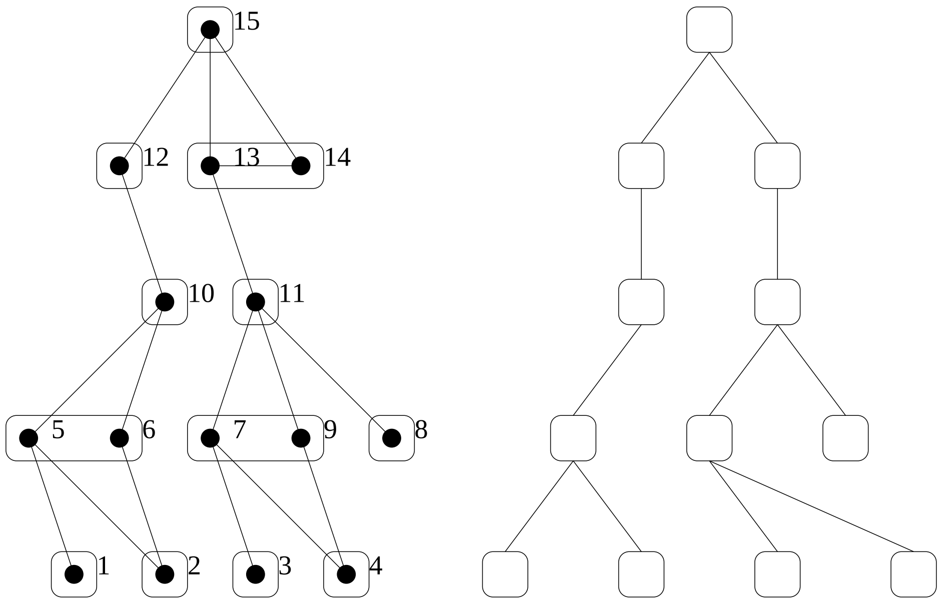
Geeignete Datenstruktur: siehe UNION/FIND-Problem.

Arbeitsweise an folgendem Beispiel (= Quotientengraph bezüglich der T_{ij})



Ausgangsbaum: Kette (Levelmengen)

ergibt



Verfeinerter Ouotientenbaum

KAPITEL 4

Unsymmetrische Matrizen

Aufgabe jetzt: Berechne $A = LDU$ für Matrix A mit

- A dünn besetzt,
- A ist nicht spd (symmetrisch indefinit oder **unsymmetrisch**).

Genauer: Bestimme zwei Permutationsmatrizen P, Q, so dass $PAQ^T = LDU$ mit P, Q, so dass

a) Faktorisierung numerisch stabil,

b) L, U besitzen wenig Fill-ins.

a) und b) können in der Regel nicht gleichzeitig optimal erreicht werden. Deshalb entfällt bei unsymmetrischen Matrizen auch die bisher gewohnte Trennung von numerischer und symbolischer Phase.
Wir kümmern uns zuerst um Fragen der numerischen Stabilität.

Abschnitt 4.1

Schwellen-Pivotwahl

Wir betrachten folgende Variante der Gauß-Elimination (Algorithmus 1.3.1) mit einem vorgegebenen Schwellenparameter $u \in (0, 1]$.

4.1.1 Algorithmus
(Gauß-Elimination mit Schwellen-Pivotwahl)

for $k = 1$ to $n - 1$ **do**
  wähle ein s_k mit $|a^{(k)}_{s_k,k}| \geq u \cdot \max_{i=k}^{n} |a^{(k)}_{ik}|$ {Schwellen-Pivotwahl}
  vertausche Zeilen s_k und k
  for $i = k + 1$ to n **do**
    $\ell_{ik} = a^{(k)}_{ik} / a^{(k)}_{kk}$
  end for
  for $i = k + 1$ to n **do**
    for $j = k + 1$ to n **do**
      $a^{(k+1)}_{ij} = a^{(k)}_{ij} - \ell_{ik} a^{(k)}_{kj}$
    end for
  end for
end for

Bemerkungen:

a) $u = 1$ „übliche“ Gauß-Elimination mit Spaltenpivotsuche.

b) Auch Algorithmus 4.1.1 bestimmt (implizit) eine Zerlegung $PA = LDU$.

c) $u < 1$: mehr Freiheit bei der Wahl des Pivot-Elements $\Rightarrow$ verwendbar zur Vermeidung von Fill-in.

Wie hängt die Stabilität von u ab?

Erinnerung an Definition 1.3.20 und Satz 1.3.19:

$$\rho_{ij} \quad := \quad \max_{k=1}^{n} |a^{(k)}_{ij}|$$

$$\rho \quad = \quad \max_{i,j=1}^{n} \rho_{ij} \qquad \text{„Wachstumsfaktor“}$$

ρ ist ein Maß für die Rückwärts-Stabilität der Gauß-Elimination (Satz 1.3.19).

4.1.2 Satz
Bei Schwellen-Pivotwahl gilt

$$\rho \leq \left(1 + \frac{1}{u}\right)^{n-1} \cdot \max_{i,j=1}^{n} |a_{ij}|.$$

Die obere Schranke wird auch angenommen.

Beweis: Übungsaufgabe. □
Für dünn besetzte Matrizen ist dieses Resultat aber viel zu pessimistisch.

4.1.3 Satz
Es sei $PA = LDU$ berechnet mit Algorithmus 4.1.1, Schwellenparameter u. Für $j = 1, \ldots, n$ sei p_j die Zahl der Nichtnullen in $U_{\cdot j}$, exlusive Diagonale, also $0 \leq p_j \leq j - 1$. Weiter sei

$$\rho_j^{(k)} = \max_{i=1}^{n} |a_{ij}^{(k)}| \qquad j, k = 1, \ldots, n.$$

Dann gilt

a) $\rho_j^{(k+1)} \begin{cases} \leq \ (1 + \frac{1}{u}\rho_j^{(k)}) & \text{falls } j \geq k+1 \text{ und } a_{sj}^{(k)} \neq 0 \ (s\text{: Pivotzeile}) \\ = \ \rho_j^{(k)} & \text{sonst,} \end{cases}$

b) $\rho_j^{(k)} \leq (1 + \frac{1}{u})^{p_j} \rho_j^{(1)}$,

c) $\rho \leq \left(\max_{j=1}^{n} \left(1 + \dfrac{1}{u}\right)^{p_j}\right) \max_{i,j=1}^{n} |a_{ij}|.$

Beweis:

a): Obere Zeile erhält man wie Satz 4.1.2, siehe also Übung. Untere Zeile: Ist $j \leq k$ oder $a_{s_j}^{(k)} = 0$, so bleibt die j-te Spalte von $A^{(k)}$ unverändert bei Transformation $A^{(k)} \rightsquigarrow A^{(k+1)}$ (bis auf die Vertauschung von zwei Zeilen), also ist $\rho_j^{(k+1)} = \rho_j^{(k)}$.

b): folgt aus a), wenn man weiß, für wieviele k bei festem j jemals $a_{sj}^{(k)} \neq 0$ ist. Da nach dem Vertauschen $a_{sj}^{(k)} \neq 0$ in Zeile k gelangt (und dann dort bleibt), ist die Anzahl dieser k gleich $p_j + 1$ (+1 für Diagonal-Element). Für das letzte solche k (=Diagonalelement) wird Spalte j nicht mehr verändert. Also

$$\rho_j^{(k)} \leq \left(1 + \frac{1}{u}\right)^{p_j} \rho_j^{(1)}.$$

c): Folgt aus b).

□

Folgerung: Für dünn besetzte Matrizen ist $p_j \ll n$; das Stabilitätsproblem also wesentlich gutartiger als bei dichten Matrizen. Für u können relativ kleine Werte genommen werden, z.B. $u = \frac{1}{100}$.

In Praxis: Satz 4.1.3 liefert nur eine Worst-Case-Abschätzung. Man sollte also z.B. während der Berechnung von $PA = LDU$ die Größe

$$\rho^{(k)} = \max_{i,j=1,\dots,n/\ell=1,\dots,k}^{n} |a_{ij}^{(\ell)}|$$

jeweils mitberechnen und aufdatieren.

Weitere Möglichkeit: Man berechnet *a posteriori*-Schranken nach folgendem Satz.

4.1.4 Satz
Es sei $PA = LDU (=: A)$. Dann gilt für $p, q \in [1, \infty]$, $\frac{1}{p} + \frac{1}{q} = 1$ $(i, j \geq k)$

$$\begin{aligned}
&(4.1.1) \qquad & |a_{ij}^{(k)}| &\leq \|(\ell_{ik}, \dots, \ell_{ii})\|_p \cdot \|(d_{kk}u_{kj}, \dots, d_{jj}u_{jj})\|_q \\
&(4.1.2) \qquad & &\leq \|L_{i\cdot}\|_p \cdot \|(DU)_{\cdot j}\|_q.
\end{aligned}$$

Beweis: (4.1.2) folgt sofort aus (4.1.1), da $\|\cdot\|_p$ nicht kleiner wird, wenn die Vektoren verlängert werden. Für (4.1.1): Für festes k partitionieren wir

$$L = [L_1|L_2] \qquad L_1 \text{ hat } k-1 \text{ Spalten}$$

$$D = \begin{bmatrix} D_1 & 0 \\ 0 & D_2 \end{bmatrix} \qquad D_1 \in \mathbb{R}^{k-1 \times k-1}$$

$$U = \begin{bmatrix} U_1 \\ U_2 \end{bmatrix} \qquad U_1 \text{ hat } k-1 \text{ Zeilen}$$

Aus $A = L(DU)$ folgt

$$A = [L_1|L_2] \begin{bmatrix} D_1U_1 \\ D_2U_2 \end{bmatrix} = L_1(D_1U_1) + L_2(D_2U_2).$$

$$\begin{aligned} \Rightarrow \qquad & A - L_1(D_1U_1) = L_2(D_2U_2). \\ \Longleftrightarrow \qquad & A^{(k)} = L_2(D_2U_2). \end{aligned}$$

Damit gilt

$$a_{ij}^{(k)} = (L_2(D_2U_2))_{ij} = (\ell_{ik}, \ldots, \ell_{in}) \begin{pmatrix} d_{kk}u_{kj} \\ \vdots \\ d_{nn}u_{nj} \end{pmatrix}.$$

Damit folgt (4.1.1) direkt aus der Hölder-Ungleichung. □

Bemerkung: Bei Schwellen-Pivotwahl ist

$$|\ell_{ik}| \leq \frac{1}{u}, \text{ also } \|L_{i\cdot}\|_\infty \leq \frac{1}{u}.$$

Nimmt man also $p = \infty$, $q = 1$, so ist zu berechnen

$$\|(DU)_{\cdot j}\|_1 \quad , j = 1, \ldots, n$$

mit Aufwand $nnz(U)$. Dann ist $\frac{1}{u} \max\limits_{j=1}^{n} \|(DU)_{\cdot j}\|_1$ a posteriori obere Schranke für ρ.

Abschnitt 4.2

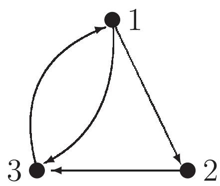
Digraphen und starke Zusammenhangskomponenten

4.2.1 Definition
Ein *Digraph* (gerichteter Graph) ist ein Paar $D = (V, E)$ mit $E \subseteq V \times V$.

$$\begin{array}{rcl} i \in V & : & \text{Knoten} \\ (i, j) \in E & : & \text{Kanten} \end{array}$$

Beachte: $(i, j) \neq (j, i)$ (während bei ungerichteten Graphen $\{i, j\} = \{j, i\}$)

4.2.2 Beispiel



4.2.3 Definition
Der zu $A \in \mathbb{R}^{n \times n}$ *gehörige Digraph* $D(A)$ ist gegeben durch $V = \{1, \ldots, n\}$, $E = \{(i, j) : a_{ij} \neq 0 \wedge i \neq j\}$.

Beispiel: Für

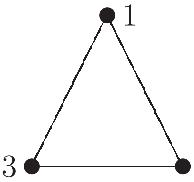
$$A = \begin{pmatrix} * & * & * \\ * & * & \\ & * & * \end{pmatrix}$$

ist $D(A)$ der Digraph aus Beispiel 4.2.2.

4.2.4 Definition
Der zu einem Digraphen $D = (V, E)$ *gehörige ungerichtete Graph* $G = |D| = (V, E')$ ist gegeben durch

$$E' = \{\{i, j\} : (i, j) \in E \text{ oder } (j, i) \in E\}.$$

Beispiel: Für D aus Beispiel 4.2.2 ist $|D|$



4.2.5 Definition

a) Die Begriffe *verbindbar*, *Kantenzug* und *Zyklus* werden für Digraphen — jetzt mit gerichteten Kanten — analog zu ungerichteten Graphen definiert. (Ausnahme: Zyklen können jetzt bereits Länge 2 haben.)

b) D heißt *zusammenhängend*, wenn $|D|$ zusammenhängend ist. Zusammenhangskomponenten entsprechend.

Beobachtung: Besitzt A die Block-Dreiecksgestalt

$$A = \begin{bmatrix} A_{11} & & 0 \\ \vdots & \ddots & \\ A_{N1} & \dots & A_{NN} \end{bmatrix}$$

mit $A_{ii} \in \mathbb{R}^{n_i \times n_i}$, $\sum_{i=1}^{n} n_i = n$, so lässt sich $Ax = b$ „blockweise" lösen durch $(x = (x_1, \dots, x_N), x_i \in \mathbb{R}^{n_i})$

for $i = 1$ to N **do**

  löse $A_{ii}x_i = b_i - \sum_{j=1}^{i-1} A_{ij}x_j$

end for

Wir benötigen also nur die LDU-Faktorisierung der Diagonalblöcke A_{ii}.

4.2.6 Definition
$A \in \mathbb{R}^{n \times n}$ heißt *irreduzibel*, wenn ($n = 1$ und $A \neq 0$) oder im zugehörigen Digraphen $D(A)$ jeder Knoten i mit jedem Knoten $j \neq i$ verbindbar ist. Andernfalls heißt A reduzibel.

4.2.7 Lemma
Die folgenden Aussagen sind äquivalent für $n \geq 2$:

a) A ist reduzibel.

b) Es existiert eine Permutationsmatrix P und $n_1 \in \{1, \ldots, n-1\}$ so, dass
$$PAP^T = \begin{pmatrix} A_{11} & 0 \\ A_{21} & A_{22} \end{pmatrix}$$
mit $A_{11} \in \mathbb{R}^{n_1 \times n_1}$, $A_{22} \in \mathbb{R}^{n_2 \times n_2}$ mit $n_2 = n - n_1$.

c) Es gibt zwei disjunkte Mengen S_1, S_2, $S_1 \neq \emptyset \neq S_2$, $S_1 \cup S_2 = \{1, \ldots, n\}$ mit $a_{ij} = 0$ falls $i \in S_1, j \in S_2$.

Beweis:

„a) $\Rightarrow$ c)“: Es seien $v \neq w$ zwei Knoten, welche in $D(A)$ nicht verbindbar sind. Es existiert also kein Kantenzug von v nach w. Setze
$$\begin{aligned} S_1 &= \{x : \text{ es ex ein Kantenzug von } v \text{ nach } x\} \cup \{v\} \\ S_2 &= \{1, \ldots, n\} \setminus S_1 \supseteq \{w\}. \end{aligned}$$
Dann gilt: $S_1 \cap S_2 = \emptyset$, $S_1 \neq \emptyset \neq S_2$, $S_1 \cup S_2 = \{1, \ldots, n\}$. Angenommen für ein $(i, j) \in S_1 \times S_2$ wäre $a_{ij} \neq 0$. Dann existiert ein Kantenzug von v über i nach j, also $j \in S_1$ im Widerspruch zu $j \in S_2$. Also ist $a_{ij} = 0$.

„c) $\Rightarrow$ b)“: Wähle die Permutation π so, dass $S_1 = \{\pi(1), \ldots, \pi(n_1)\}$, $S_2 = \{\pi(n_1+1), \ldots, \pi(n)\}$. Sei P die zugehörige Permutationsmatrix. Dann gilt: $(PAP^T)_{ij} = a_{\pi(i)\pi(j)}$ und für $i \in \{1, \ldots, n_1\}, j \in \{n_1 + 1, \ldots, n\}$ ist $\pi(i) \in S_1, \pi(j) \in S_2$, also $a_{\pi(i)\pi(j)} = 0$.

„b) $\Rightarrow$ a)“: Sei π die zu P gehörige Permutation. In $D(PAP^T)$ gibt es keinen (gerichteten) Kantenzug von 1 nach n. Also gibt es in $D(A)$ keinen Kantenzug von $\pi^{-1}(1)$ nach $\pi^{-1}(n)$. Also ist A reduzibel.

□

4.2.8 Definition
Die *reduzible Normalform* einer Matrix A ist eine Darstellung der Gestalt

$$PAP^T = \begin{bmatrix} A_{11} & & 0 \\ \vdots & \ddots & \\ A_{N1} & \ldots & A_{NN} \end{bmatrix} \tag{4.2.1}$$

wobei $A_{ii} \in \mathbb{R}^{n_i \times n_i}$ entweder irreduzibel ist, oder ein 1×1-Block mit Wert 0; P ist Permutationsmatrix.

Uns interessieren Algorithmen zur Bestimmung von P in (4.2.1). Dies stellt sich als ein „Problem auf dem Digraphen $D(A)$“ heraus.

4.2.9 Definition
$D = (V, E)$ sei ein Digraph. $v, w \in V$ heißen *stark verbunden* in D, falls $v = w$ oder (es existieren Kantenzüge von v nach w und von w nach v).

4.2.10 Satz
„stark verbunden“ ist eine Äquivalenzrelation auf V.

Beweis: Siehe Übung. □

Folgerung: V wird durch die zugehörigen Äquivalenzklassen C_k partitioniert. Die C_k heißen *starke Zusammenhangskomponenten*.

Folgerung: $A \in \mathbb{R}^{n\times n}$, $n \geq 2$, ist irreduzibel, genau dann wenn $D(A)$ genau eine starke Zusammenhangskomponente besitzt.

4.2.11 Definition
Der *Quotientengraph* $\mathcal{D}$ eines Digraphen bezüglich einer Partition

$$\mathcal{V} = \{V_1, \ldots, V_k\} \text{ mit } V = \bigcup_{i=1}^{k} V_i$$

ist gegeben durch $\mathcal{D} = (\mathcal{V}, \mathcal{E})$ mit

$$\mathcal{E} = \{(V_i, V_j) : \exists\, v \in V_i, w \in V_j \text{ mit } (v, w) \in E\}.$$

4.2.12 Lemma
Die folgenden Aussagen für einen Digraphen D sind äquivalent:

a) C ist eine starke Komponente von D.

b) Für alle $v \in C$ gilt: existiert ein Zyklus, der v und $w \neq v$ enthält, so gilt $w \in C$.

c) Es existiert ein $v \in C$ mit

$$C = \{w \in V : \text{ es ex Zyklus, der } v \text{ und } w \text{ enthält}\} \cup \{v\}.$$

Beweis: Übungsaufgabe. □

4.2.13 Satz
Der Quotientengraph $\mathcal{D}$ von D bzgl. der starken Komponenten ist azyklisch, d.h. er enthält keinen (gerichteten) Zyklus.

Beweis: Angenommen, es existiert ein Zyklus $(i_1, i_2, \dots, i_\ell)$ mit $i_1 = i_\ell$, $\ell \geq 3$ in $\mathcal{D}$, also $(i_\nu, i_{\nu+1}) \in \mathcal{E}$ für $\nu = 1, \dots, \ell - 1$. O.B.d.A. sind bis auf $i_1 = i_\ell$ alle Knoten verschieden; wir nummerieren o.B.d.A. $i_1 = 1, \dots, i_{\ell-1} = \ell - 1$, $i_\ell = 1$. Es existieren also in $D = (V, E)$ Knoten $v_1, \dots, v_{\ell-1}$ und $w_2, \dots, w_\ell$ mit $v_i, w_i \in C_i$, $i = 1, \dots, \ell-1$, $w_\ell \in C_1$ und $(v_i, w_{i+1}) \in E, i = 1, \dots, \ell-1$. Da C_i starke Komponente ist, existiert ein Weg von w_i nach v_i, $i = 2, \dots, \ell - 1$ und von w_ℓ nach v_1. Durch Zusammensetzen erhält man also einen Zyklus, welcher alle v_i und w_i enthält. Damit folgt nach Lemma 4.2.12 (b) insbesondere $v_2 \in C_1$. Widerspruch! $\square$

4.2.14 Satz
Jeder azyklische Digraph kann topologisch sortiert werden, d.h. es existiert eine Permutation π auf V, so dass gilt

$$(i, j) \in E \Rightarrow \pi(i) < \pi(j).$$

Beweis: Siehe Info II (Knoten mit Eingangsgrad 0 nach vorne sortieren; Aufwand $\mathcal{O}(|E| + |V|)$). $\square$

4.2.15 Satz
$A \in \mathbb{R}^{n \times n}$ besitzt die reduzible Normalform

$$PAP^T = \begin{bmatrix} A_{11} & & 0 \\ \vdots & \ddots & \\ A_{N1} & \dots & A_{NN} \end{bmatrix}$$

mit $A_{ii} \in \mathbb{R}^{n_i \times n_i}$, A_{ii} irreduzibel oder $(n_i = 1, A_{ii} = 0)$ genau dann, wenn für die zu P gehörige Permutation π gilt

a) die starken Komponenten von $D(A)$ sind

$$C_k = \{\pi(m_{k-1} + 1), \dots, \pi(m_k)\}, k = 1, \dots, N, m_k = \sum_{i=1}^{k} n_i,$$

b) π sortiert die C_k umgekehrt topologisch, d.h. für alle $i, j \in \{1, \dots, n\}$ gilt $\pi(i) \in C_k, \pi(j) \in C_{k'}$ und $a_{ij} \neq 0 \Rightarrow k' \leq k$.

Beweis: Wir verwenden:

$$(4.2.2) \qquad \begin{array}{ll} & A \text{ irreduzibel oder } (A \in \mathbb{R},\ A = 0) \\ \iff & D(A) \text{ besitzt genau eine starke Komponente (trivial).} \end{array}$$

„$\Leftarrow$“ Aus a) und (4.2.2) folgt: Für $k = 1, \ldots, N$ sind die Blöcke

$$\begin{aligned}(PAP^T)_{i,j=m_{k-1}+1,\ldots,m_k} &= (a_{\pi(i),\pi(j)})_{i,j=m_{k-1}+1,\ldots,m_k}\\ &= (a_{\mu\nu})_{\mu,\nu\in C_k}\end{aligned}$$

irreduzibel, oder $= 0 \in \mathbb{R}$. In der Blockdarstellung

$$PAP^T = \begin{bmatrix} A_{11} & \ldots & A_{1N} \\ \vdots & & \vdots \\ A_{N1} & \ldots & A_{NN} \end{bmatrix}$$

sind die Diagonalblöcke dann irreduzibel oder $= 0 \in \mathbb{R}$. Zu zeigen bleibt $A_{kk'} = 0$ für $k' > k$. Angenommen, für $k' > k$ ist $A_{kk'} \neq 0$. Dann existieren $\pi(i) \in C_k, \pi(j) \in C_{k'}$ mit $a_{\pi(i)\pi(j)} \neq 0$. Nach b) ist aber dann $k' \leq k$. Widerspruch!

„$\Rightarrow$“ Geht genauso.

□

Vorgehensweise zur Bestimmung der reduziblen Normalform damit

1) Bestimme die starken Komponenten von $D(A)$.

2) Sortiere den zugehörigen Quotientengraphen topologisch.

3) Bestimme eine Permutation, welche die topologisch sortierten Komponenten der Reihe nach nummeriert, d.h. ist $C_k = \{i_{1,k}, \ldots, i_{n_k,k}\}, k = 1, \ldots, N$ (bereits topologisch sortiert), so ist

$$\begin{aligned}\pi : 1 &\rightarrow i_{1,1}\\ &\vdots\\ n_1 &\rightarrow i_{n_1,1}\\ n_1+1 &\rightarrow i_{1,2}\\ &\vdots\\ n_1+n_2 &\rightarrow i_{n_2,2}\\ &\vdots\end{aligned}$$

Uns interessiert jetzt Teil 1).

Zur Vorbereitung benötigen wir eine „Verfeinerung“ der Depth-first-Suche in einem Digraphen, bei welcher die erreichbaren Knoten zwei Mal nummeriert werden:

- bei erstmaliger Entdeckung: $a(v)$,
- bei Beendigung der depth-first-Suche ausgehend von diesem Knoten: $e(v)$.

4.2.16 Algorithmus
dfs(s)

{depth-first search in Digraphen, ausgehend von Knoten s. Für alle entdeckten Knoten v werden Nummern $a(v)$ und $e(v)$ vergeben}

for all $v \in V$ **do**
  $a(v) := 0; e(v) := 0$
end for
$k := 0; \ell := 0$
k++, $a(s) := k$
dfr(k, ℓ, a, e, s)
ℓ++; $e(s) := \ell$

wobei

4.2.17 Algorithmus
dfr(k, ℓ, a, e, v)

{rekursiver Teilalgorithmus für dfs; v ist aktueller, zu untersuchender Knoten; a, e sind Vektoren mit bisher vergebenen Nummern; k, ℓ sind zuletzt vergebene Nummern}

for all von v ausgehenden Kanten (v, w) **do**
  if $a(w) = 0$ **then**
    k++; $a(w) = k$
    dfr(k, ℓ, a, e, w)
    ℓ++; $e(w) = \ell$
  end if
end for

Wir hatten früher eine nicht-rekursive Beschreibung der depth-first Suche kennengelernt unter Verwendung der Datenstruktur Stack. Diese ist mit der rekursiven Variante *nicht* identisch, wohl aber der folgende Algorithmus.

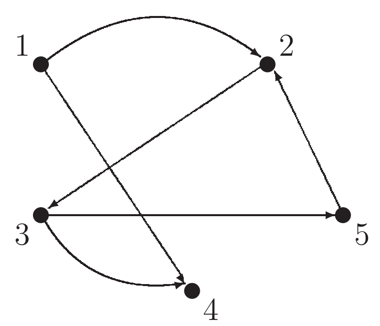
4.2.18 Algorithmus
Depth-First mit Nummern, nicht rekursiv

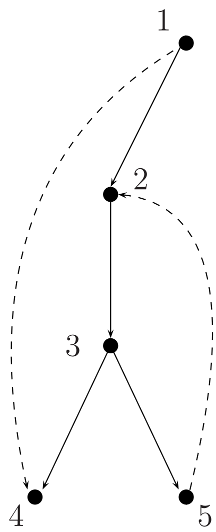
{zu dfs(s) äquivalenter Algorithmus, verwendet einen Stack}

$k := 0$, $\ell := 0$
$k++$, $a(s) := k$
füge s in leeren Stack S ein
while $S \neq \emptyset$ **do**
  $v :=$ oberstes Element von S
  if es ex. nicht markiertes w mit $(v, w) \in E$ **then**
    wähle ein solches w
    $k++$, $a(w) := k$ {jetzt gilt w als markiert}
    füge w in S ein
  else
    entnehme v aus S
    $\ell++$, $e(v) := \ell$
  end if
end while

Wichtige Folgerung: Gilt $a(w) < a(v)$ **und** $e(w) > e(v)$, so existiert ein Weg von w nach v. (Der Stack S stellt immer einen Weg von seinem untersten zum obersten Element dar.)

Beispiel





4.2.19 Bemerkung

a) Algorithmus 4.2.16 bzw. 4.2.18 „entdeckt“ nur die Knoten des Teilgraphen $\tilde{D}$, dessen Knoten von v aus erreichbar sind. Sei $\tilde{D} = (\tilde{V}, \tilde{E})$, so gilt $|\tilde{E}| \geq |\tilde{V} - 1|$, denn jeder Knoten $w \in \tilde{V}, w \neq v$ ist Endpunkt mindestens einer Kante.

b) Der Aufwand für Algorithmus 4.2.17 ist $\mathcal{O}(|\tilde{E}|)$, denn jede Kante wird genau einmal betrachtet mit Aufwand $\mathcal{O}(1)$.

c) Der Aufwand für Algorithmus 4.2.16 und 4.2.18 ist $\mathcal{O}(|\tilde{E}| + |V|)$.

4.2.20 Definition
Ein Knoten s eines Digraphen heißt *Wurzel*, falls ein Kantenzug von s zu jedem Knoten $v \neq s$ existiert.

4.2.21 Definition
Sei $D = (V, E)$ ein Digraph. Dann ist $D^T = (V, E^T)$ mit $E^T = \{(i, j) : (j, i) \in E\}$ der zugehörige *transponierte Digraph*.

Der folgende Algorithmus von Aho, Hopcroft, Ullman (1983) bestimmt die starken Komponenten in einem Digraphen mit Wurzel s.

4.2.22 Algorithmus

{ bestimmt die starken Komponenten im Digraphen D mit Wurzel s}

$k = 1$
bestimme mit dfs(s) die Endnummern $e(v)$ für D
setze $H = D^T = (V, E^T)$
while $V \neq \emptyset$ **do**
  wähle r aus V mit $e(r)$ maximal
  bestimme Anfangsnummern $a'(w)$ auf H mittles dfs(r)
  setze $C_k = \{w \in V : a'(w) \neq 0\}$
  setze $V = V \setminus C_k$, $H = H|_V$
  $k = k + 1$
end while

4.2.23 Satz
Algorithmus 4.2.22 berechnet die starken Komponenten eines Digraphen D mit Wurzel s. Der Aufwand ist $\mathcal{O}(|E|)$.

Beweis: Aufwand: Nach Bemerkung 4.2.19 ist $|V| = \mathcal{O}(|E|)$. Einmalige Anwendung von dfs(s) auf D hat Aufwand: $\mathcal{O}(|V| + |E|) = \mathcal{O}(|E|)$.
dfs(s) in Schleife: Aufwand jeweils $\mathcal{O}(|E_r|)$, wobei E_r Menge der von r aus zugänglichen Kanten in H, also

$$\bigcup_{\text{„}r\,\in\,\text{Schleife“}} E_r \subseteq E^T; E_r \text{ paarweise disjunkt.}$$

Es ist

$$\sum_{\text{„}r\,\in\text{Schleife“}} |E_r| \leq |E^T| = |E|.$$

Gesamtaufwand also $\mathcal{O}(|E|)$.
Korrektheit: Wir zeigen, dass C_1 starke Komponente von D ist. Der Rest folgt dann durch Induktion. Dazu zeigen wir:

(i) Existiert im Digraphen D ein Weg von v nach r, so ist $v \in C_1$.

(ii) Ist $v \in C_1$, so existiert ein Weg von v nach r und von r nach v in D.

zu (i): Wenn in D ein Weg von v nach r existiert, gibt es in D einen von r nach v. Bei dfs(r) wird $a'(v)$ also geändert, d.h. v kommt in C_1.

zu (ii): $v \neq r \in C_1 \Rightarrow$ es existiert ein Weg von r nach v in $H \Rightarrow$ es existiert ein Weg von v nach r in D. Sei $a()$ die Anfangsnummer von dfs auf D (!). Angenommen, es wäre $a(r) > a(v)$. Da in D ein Weg von v nach r existiert, folgt $e(r) < e(v)$ im Widerspruch zur Wahl von r. Also gilt $a(r) < a(v)$ (und $e(r) > e(v)$). Es existiert in D also ein Weg von r nach v.

Für den Induktionsschritt muss man noch überlegen, dass die starken Komponenten von D nach Entfernung von C_1 gerade die übrigen starken Komponenten sind. $\rightsquigarrow$ Lemma 4.2.26(i).

$\square$

Um alle starken Komponenten eines Digraphen zu bekommen, ist 4.2.22 mehrfach anzuwenden.

4.2.24 Definition
$D = (V, E)$ sei ein Digraph, $s \in V$. Dann ist

$$V(s) = \{v \in V : \text{ es ex in } D \text{ ein Weg von } s \text{ nach } v\}.$$

s ist also Wurzel von $D|_{V(s)}$.

4.2.25 Algorithmus

{bestimmt alle starken Komponenten eines Digraphen}

for all Knoten v **do**
  $a(v) = 0; e(v) = 0;$
end for
while Knoten s existiert mit $a(s) = 0$ **do**
  wähle einen solchen Knoten
  bestimme die starken Komponenten von $V(s)$ mit Algorithmus 3
    {dabei wird D verändert}
end while

Die Korrektheit dieses Algorithmus beweist folgendes Lemma.

4.2.26 Lemma
Sei $D = (V, E)$ ein Digraph, $C_1, \ldots, C_\ell$ seine starken Komponenten und $s \in V$. Dann gilt für $k = 1, \ldots, \ell$:

(i) Die starken Komponenten von $D|_{C_2 \cup \ldots \cup C_\ell}$ sind $C_2, \ldots, C_\ell$.

(ii) $C_k \subseteq V(s)$ oder $C_k \cap V(s) = \emptyset$.

(iii) Im Fall $C_k \subseteq V(s)$ ist C_k auch starke Komponente von $D|_{V(s)}$.

Beweis:

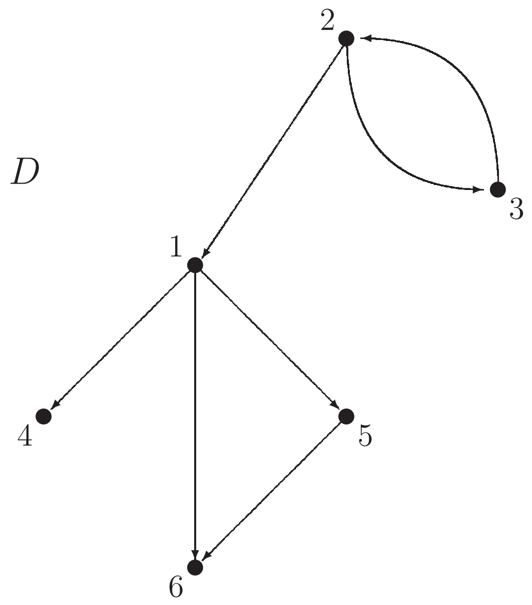
(i): Übungsaufgabe.

(ii): Übungsaufgabe.

(iii): $C_k \subseteq V(s) \Rightarrow D|_{C_k} \subseteq D|_{V(s)}$. Also ist C_k auch in $D|_{V(s)}$ stark zusammenhängend. Gäbe es einen Knoten $w \in V(s), w \notin C_k$, welcher mit einem Knoten $v \in C_k$ stark verbunden wäre in $D|_{V(s)}$, so wäre w mit v auch in D stark verbunden. Also wäre $w \in C_k$, Widerspruch!

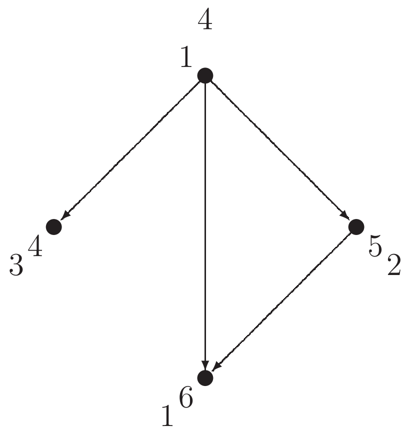
$\square$

Beispiel



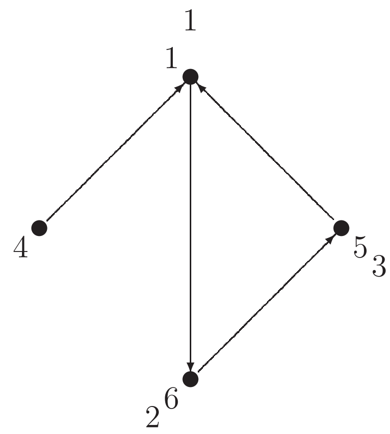
Wir starten mit $s = 1$, $V(1) = \{1, 5, 6, 4\}$, Teilgraph.

$$D_1 = D|_{V(1)}$$



$e(v)$ auf D_1

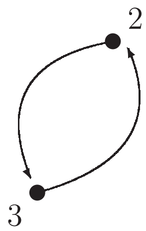
D_1^T



$C_1 = \{1, 5, 6\}$, $C_2 = \{4\}$.

Jetzt nehmen wir z.B. $s = 3$

$D = D|_{V(3)}$



$D^T = D$

$C_3 = \{2, 3\}$

Abschnitt 4.3

Strukturelle Singularität und Transversale

4.3.1 Definition
$A \in \mathbb{R}^{n\times n}$ heißt *strukturell singulär*, falls gilt:
Für alle A' mit $struct(A') = struct(A)$ ist A' singulär.

4.3.2 Beispiel

$$A = \begin{bmatrix} * & * & \\ & & * \\ & & * \end{bmatrix} \text{ ist strukturell singulär.}$$

4.3.3 Lemma
$A \in \mathbb{R}^{n\times n}$ ist nicht strukturell singulär $\Longleftrightarrow$ es existiert eine Permutationsmatrix P mit $(PA)_{ii} \neq 0$, $i = 1, \dots, n$.

Beweis:

„$\Leftarrow$“ Wir setzen

$$B_{ij} = \begin{cases} n & \text{für } i = j \\ 1 & \text{für } i \neq j, (i,j) \in struct(PA) \\ 0 & \text{sonst.} \end{cases}$$

Dann ist B streng diagonaldominant, insbesondere regulär. Setze $A' = P^{-1}A$. Dann ist A' regulär und $struct(A') = struct(A)$.

„$\Rightarrow$“: Folgt aus Satz 4.3.8. (Korrektheitsbeweis zu Algorithmus 4.3.6, siehe unten.)

□

4.3.4 Definition
Eine Permutation π heißt *Transversale* von A, falls

$$A_{\pi(i),i} \neq 0, \; i = 1, \dots, n$$

gilt (also gilt $(PA)_{ii} \neq 0$, wenn P die zugehörige Permutationsmatrix ist).

Strukturell singuläre Matrizen sind also solche ohne Transversale.

Idee zur Entwicklung eines Algorithmus, der Transversale berechnet, bzw. erkennt, dass keine existiert:
In Schritt k sei Permutation π_k bekannt mit $a_{\pi_k(i),i} \neq 0$, $i = 1, \dots, k-1$. Zur Vereinfachung der Notation sei $A \leftarrow PA$, also $a_{ii} \neq 0$, $i = 1, \dots, k-1$.

Jetzt suchen wir Permutation σ_k mit $a_{\sigma_k(i),i} \neq 0$, $i = 1, \dots, k$.

Einfacher Fall: $\text{struct}(A(k:n,k)) \neq \emptyset$ (σ_k vertauscht j und k)

Schwieriger Fall: $\text{struct}(A(k:n,k)) = \emptyset$ (σ_k ist als Zykel zu bestimmen, s.u.)

4.3.5 Definition
(i,j) bezeichne die „Transposition" von i und j, also die Permutation π mit

$$\begin{aligned} \pi(i) &= j, \\ \pi(j) &= i, \\ \pi(k) &= k \text{ falls } k \neq i, j. \end{aligned}$$

(k,k) oder (k) bezeichnet beides die Identität. Eine Permutation der Gestalt

$$\sigma = (i_\ell, i_{\ell-1}) \circ (i_{\ell-1}, i_{\ell-2}) \circ \dots \circ (i_2, i_1)$$

heißt *Zykel* (der Länge ℓ), abgekürzt

$$\sigma = (i_\ell, i_{\ell-1}, \dots, i_1).$$

Falls die i_ν paarweise verschieden sind, gilt dann

$$\begin{aligned} \sigma(i_1) &= i_\ell \\ \sigma(i_2) &= i_1 \\ &\vdots \\ \sigma(i_\ell) &= i_{\ell-1} \end{aligned}$$

Wir versuchen, einen Zykel σ_k der Gestalt

$$\text{(4.3.1)} \quad \begin{cases} \sigma_k = (i_1, \dots, i_\ell) \text{ mit } i_1 = k; i_\nu < k, \nu = 2, \dots, \ell-1; i_\ell \geq k \text{ mit} \\ i_\nu, 1 \leq \nu \leq \ell \text{ paarweise verschieden} \\ a_{i_{\nu+1}, i_\nu} \neq 0, \quad \nu = 1, \dots, \ell-1 \\ a_{i_1, i_\ell} \neq 0 \end{cases}$$

zu finden. Dann gilt nämlich

$$\begin{aligned} i \in \{1, \dots, k\} \setminus \{i_1, \dots, i_\ell\} \quad & a_{\sigma_k(i),i} = a_{ii} \neq 0 \\ i = i_\nu, \nu \in \{1, \dots, \ell\} \quad & a_{\sigma_k(i_\nu), i_\nu} = a_{i_{\nu+1}, i_\nu} \neq 0. \end{aligned}$$

4.3.6 Algorithmus
(Algorithmus von Hall)
fortsetzen(σ, k)

{$\sigma = (i_1, \ldots, i_\ell)$ ist Zykel mit $i_1 = k$, $i_\nu < k$, $\nu = 2, \ldots, \ell$, alle verschieden, $a_{i_{\nu+1}, i_\nu} \neq 0$, $\nu = 1, \ldots, \ell - 1$. Der Zykel wird jetzt um eins verlängert; nach Möglichkeit wird (4.3.1) erreicht.}

if existiert $j \geq k$ mit $a_{j,i_\ell} \neq 0$ **then** {look-ahead}
  setze $\sigma = (i_1, \ldots, i_\ell, j)$ im Fall $j \neq k$
  bzw. $\sigma = (i_1, \ldots, i_\ell)$ im Fall $j = k$ und **stoppe** { Zykel gefunden}
else {depth-first-Suche}
  for all j mit $a_{j,i_\ell} \neq 0$ **do** {$j < k$!}
    if j nicht markiert **then**
      markiere j
      fortsetzen($(i_1, \ldots, i_\ell, j), k$)
    end if
  end for
end if

4.3.7 Algorithmus
zykel(k)

{startet Algorithmus 4.3.6}
markiere k
$\sigma = (k)$ { Identität }
fortsetzen(σ, k)
if $\sigma = (k)$ **then**
  melde „Zykel existiert nicht“
else
  gebe Zykel σ aus
end if

Beispiel für den Ablauf des Algorithmus:

$$
\begin{array}{c|cccc|c}
 & 1 & 2 & 3 & 4 & 5 \\
1 & \mathrm{x} & \mathrm{x} & & & \\
2 & & \mathrm{x} & & & \mathrm{x} \\
3 & & & \mathrm{x} & \mathrm{x} & \mathrm{x} \\
4 & & & \mathrm{x} & \mathrm{x} & \mathrm{x} \\
\hline
5 & & & & & 0 \\
6 & \mathrm{x} & & & & \vdots
\end{array}
$$

σ	markiert wird
(5)	5
(5 4)	4
(5 4 3)	3
(5 4)	
(5)	
(5 2)	2
(5 2 1)	1
(5 2 1 6)	

Der gesuchte Zykel ist (5216); nach Permutation haben wir

$$
\begin{array}{cccccc}
 & 1 & 2 & 3 & 4 & 5 \\
1 & \mathrm{x} & & & & \\
2 & \mathrm{x} & \mathrm{x} & & & \\
3 & & & \mathrm{x} & \mathrm{x} & \mathrm{x} \\
4 & & & \mathrm{x} & \mathrm{x} & \mathrm{x} \\
5 & & \mathrm{x} & & & \mathrm{x} \\
6 & & & & &
\end{array}
$$

4.3.8 Satz
Endet $zyke\ell(k)$ mit der Ausgabe „Zykel existiert nicht“, so ist A strukturell singulär.

Beweis: Für diesen Fall bezeichne K die Menge aller im Depth-first-Teil markierten Indizes j. Es gilt

$$j \in K \Rightarrow j < k.$$

Im depth-first-Teil werden für jedes $i \in K$ alle j mit $a_{ij} \neq 0$ markiert (und alle sind $< k$). Sei $K = \{j_1, \ldots, j_p\}$. Die p Spalten $j_1, \ldots, j_p$ von A besitzen also Nichtnulleinträge höchstens in den Zeilen $j_1, \ldots, j_p$. Dasselbe gilt auch für die Spalte k. Also haben wir $p+1$ Spalten, welche nur in p Zeilen besetzt sind; sie sind also für jede Belegung ihrer Struktur linear abhängig. □

Folgerung Ist A nicht strukturell singulär, so besitzt A eine Transversale und der folgende Algorithmus berechnet sie.

4.3.9 Algorithmus
(Transversale)

```
P = I, π_0 = id
for k = 1 to n do
  if a_kk = 0 then
    zykel(k)                                  {liefert σ_k oder Meldung}
    if Meldung „Zykel existiert nicht“ then
      stoppe                                  {Transversale existiert nicht}
    else
      Q_k = zu σ_k^{-1} gehörige Permutationsmatrix
      A = Q_k A; P = Q_k P
    end if
  end if
end for
```

Aufwandsanalyse
Aufwand von Algorithmus 4.3.7 / Algorithmus 4.3.6 ist über alle k zu summieren (jedenfalls wenn in Algorithmus 4.3.9 statt $A \leftarrow Q_k A$, $P \leftarrow Q_k P$ implizit vertauscht wird).
Aufwand von Algorithmus 4.3.6 hat zwei Anteile:

1) Depth-First. Im worst-case werden alle Nichtnulleinträge von $A(1:k, 1:k)$ betrachtet. Aufwand also $\mathcal{O}(nnz(A(1:k,1:k)))$.

2) Look-ahead. Angenommen, Spaltenstruktur ist über gepacktes Format oder Listenformat zugänglich.

   Worst-case: Jeder look-ahead-Schritt durchläuft die ganze (besetzte) Spalte i_ℓ, $i_\ell \in \{1,\dots,k\}$. Aufwand also $\mathcal{O}(nnz(A(1:n,1:k)))$.

Gesamtaufwand:

$$\sum_{k=1}^{n} \mathcal{O}(nnz(A(1:k,1:k))) + \mathcal{O}(nnz(A(1:n,1:k))) = \mathcal{O}(n \cdot nnz(A)).$$

Da im Allgemeinen $nnz(A) = \Omega(n)$ ist dies im Prinzip eine potenzielle $\mathcal{O}(n^2)$-Falle!

Aber: Wir haben großzügig abgeschätzt. Für Anteil 2 kann der Aufwand mit Hilfe von Zeigern (siehe Duff, Erisman, Reid) auf $\mathcal{O}(nnz(A))$ reduziert werden.
Fazit: Algorithmus 4.3.9 hat sich in der Praxis bewährt.

Wir behandeln als Alternative jetzt den Algorithmus von Hopcroft und Karp zur Bestimmung einer Transversale. Er hat eine theoretisch bessere Komplexität als das Verfahren von Hall.

Der neue Algorithmus verwendet „maximale Matchings“.

4.3.10 Definition
Sei $G = (V, E)$ ein Graph.

(i) Eine Menge $M \subseteq E$ von Kanten heißt *Matching*, falls jeder Knoten mit höchstens einer Kante aus M inzident ist.

(ii) Knoten, die mit keiner Kante des Matchings M inzident sind, heißen *frei*.

(iii) Ein Matching M heißt *maximal*, wenn es kein Matching mit mehr Elementen gibt.

4.3.11 Definition
Ein Graph $P = (V, E)$ heißt *bipartit* oder *paar*, wenn es eine Partition $V = B \cup G$ gibt, so dass alle Kanten nur zwischen B und G verlaufen, $E \subseteq \{\{b, g\} \mid b \in B, g \in G\}$.

B: 'boys', G: 'girls'.

4.3.12 Definition
Für $A \in \mathbb{R}^{n\times n}$ ist $P(A) = (V, E)$ der bipartite Graph mit

$$B = \{z_1, \dots, z_n\},\ G = \{s_1, \dots, s_n\},\ V = \{\{z_i, s_j\} \mid a_{ij} \neq 0\}.$$

Abbildung 4.1 zeigt ein Beispiel.

Folgerung: Die Matrix A besitzt eine Transversale, wenn $P(A)$ ein maximales Matching der Größe n besitzt (man sagt auch: $P(A)$ ist ein *Heiratsgraph*). Denn ist $M = \{\{z_{i_1}, s_{j_1}\}, \dots, \{z_{i_n}, s_{j_n}\}\}$ ein solches Matching, und sind π_z und π_s die Permutationen mit $\pi_s(\ell) = i_\ell,\ \pi_z(\ell) = j_\ell$ sowie P_z und P_s die zugehörigen Permutationsmatrizen, so besitzt $P_z A P_s^T$ eine vollbesetzte Diagonale.

Es geht also darum, in $P(A)$ ein maximales Matching zu bestimmen (und zu zeigen, dass es die Größe n hat).

$$A = \begin{pmatrix} 0 & x & 0 & 0 & 0 \\ 0 & x & 0 & x & 0 \\ x & 0 & 0 & 0 & x \\ 0 & 0 & x & x & 0 \\ 0 & 0 & x & 0 & 0 \end{pmatrix}$$

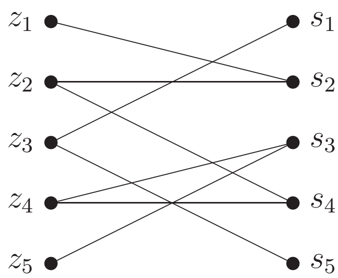


Abbildung 4.1: Matrix und zugehöriger bipartiter Graph

4.3.13 Definition
Sei M ein Matching von $G = (V, E)$ (nicht notwendig bipartit). Ein Pfad $p = (v_1, \ldots, v_{2k})$ in G heißt *zunehmender Pfad bzgl.* M, falls gilt

(i) v_1 und v_{2k} sind freie Knoten,

(ii) genau jede zweite Kante von p liegt in M. Betrachtet man p als Kantenmenge, so bedeutet das

$$p \cap M = \{\{v_2, v_3\}, \{v_4, v_5\}, \ldots, \{v_{2k-2}, v_{2k-1}\}\}$$

Zunehmende Pfade haben immer ungerade Länge.
Mit $A \oplus B$ bezeichnen wir die symmetrische Differenz zweier Mengen:

$$A \oplus B = (A \setminus B) \cup (B \setminus A) = (A \cup B) \setminus (A \cap B).$$

4.3.14 Lemma
Es seien $A, B, C \subseteq Y$ Teilmengen von Y und $A^c = \{x \in Y \mid x \notin A\}$ das Komplement von A. Dann gilt

- $A \setminus B = A \cap B^c$ und damit $A \oplus B = (A \cap B^c) \cup (A^c \cap B)$.
- $A \oplus (B \oplus C) = (A \oplus B) \oplus C$.
- $A \oplus (A \oplus B) = B$.

Beweis: Übung.

Das folgende Resultat ist klar, s. Abbildung 4.2.

4.3.15 Lemma
Ist M ein Matching und p ein zunehmender Pfad bzgl. M, so ist $M \oplus p$ ein Matching mit $|M \oplus p| = |M| + 1$.

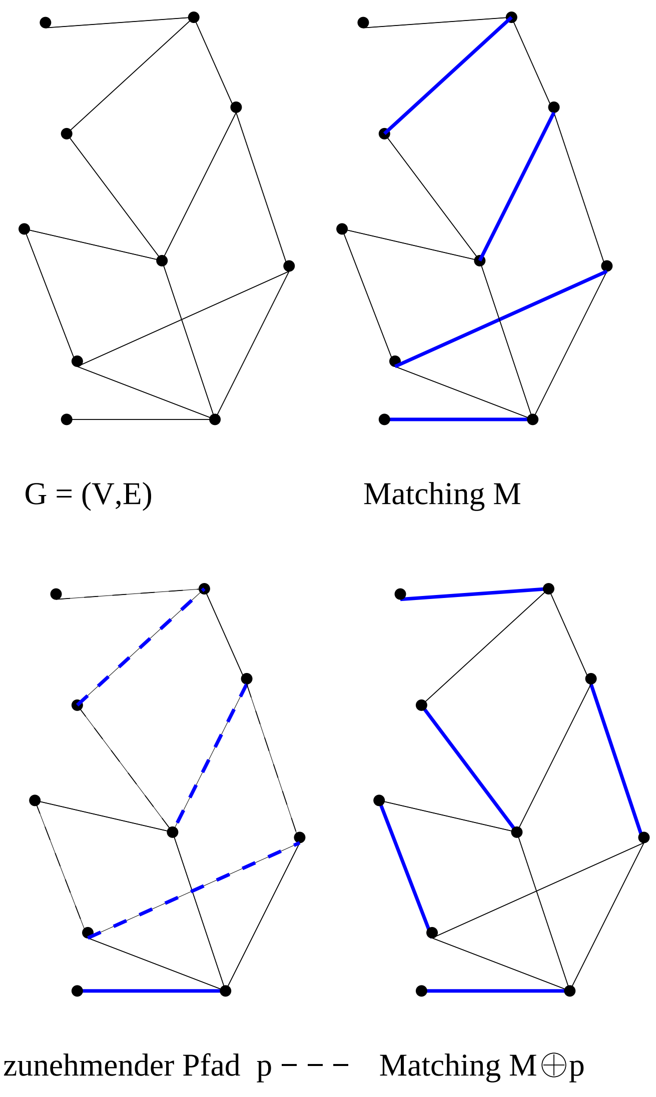


Abbildung 4.2: Matching und zunehmende Pfade

4.3.16 Satz
M und N seien Matchings mit $|M| = r < |N| = s$. Dann enthält $M \oplus N$ mindestens $s - r$ knotendisjunkte zunehmende Pfade bzgl. M.

Beweis: Betrachte den Graphen $H = (V, M \oplus N)$. In H ist jeder Knoten höchstens mit einer Kante aus $M \setminus N$ und einer aus $N \setminus M$ inzident. Für jede Zusammenhangskomponente $C_1, \ldots C_g$ (kurz: C) von H gilt deshalb einer der drei folgenden Fälle:

- C ist ein (in H) isolierter Knoten
- C ist ein Zyklus gerader Länge mit Kanten abwechselnd in $M \setminus N$ und $N \setminus M$
- C ist ein Pfad mit Kanten abwechselnd in $M \setminus N$ und $N \setminus M$

Sei $C_i = (V_i, E_i)$, setze $\delta_i = |E_i \cap N| - |E_i \cap M|$. Dann ist $\delta_i \in \{-1, 0, 1\}$, und $\delta_i = 1$ genau dann, wenn E_i ein zunehmender Pfad bzgl. M ist. Außerdem gilt

$$\begin{aligned}\sum_{i=1}^{g} \delta_i &= \sum_{i=1}^{g} |E_i \cap N| - \sum_{i=1}^{g} |E_i \cap M| \\ &= |(M \oplus N) \cap N| - |(M \oplus N) \cap M| \\ &= |N \setminus M| - |M \setminus N| = |M| - |N| = s - r.\end{aligned}$$

Also ist $\delta_i = +1$ für mindestens $s - r$ verschiedene Werte von i. Die zugehörigen V_i sind als Knotenmengen von Zusammenhangskomponenten disjunkt. □

4.3.17 Korollar
M ist ein maximales Matching genau dann, wenn es keinen zunehmenden Pfad bezüglich M gibt.

4.3.18 Korollar
Ist M ein Matching mit $|M| = r$ und hat ein maximales Matching die Größe $s > r$, so existiert ein zunehmender Pfad bzgl. M mit höchstens der Länge $2 \cdot \lfloor r/(s - r) \rfloor + 1$.

Beweis: N sei ein maximales Matching, $|N| = s$. Nach Satz 4.3.16 enthält $M \oplus N$ mindestens $s - r$ knotendisjunkte (und damit kantendisjunkte) zunehmende Pfade bzgl. M. Alle solchen Pfade zusammen enthalten höchstens r Kanten aus M, also enthält einer von ihnen höchstens $\lfloor r/(s - r) \rfloor$ Kanten

aus M. Dieser Pfad hat eine Gesamtlänge von höchstens $2 \cdot \lfloor r/(s-r) \rfloor + 1$.
$\square$

Uns interessieren die Längen kürzester zunehmender Pfade.

4.3.19 Satz
M sei ein Matching und p ein *kürzester* zunehmender Pfad bzgl. M und p' ein zunehmender Pfad bzgl. $M \oplus p$. Dann ist

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

Beweis: Setze $N = M \oplus p \oplus p'$. N ist Matching mit $|N| = |M| + 2$. Nach Satz 4.3.16 gibt es in $M \oplus N$ zwei knotendisjunkte zunehmende Pfade p_1, p_2 bzgl. M. Nach Lemma 4.3.14 ist $M \oplus N = p \oplus p'$ und damit

$$|p \oplus p'| \geq |p_1| + |p_2| \geq 2|p| \quad \text{(da } p \text{ kürzester Pfad).} \tag{4.3.2}$$

Andererseits gilt für jede symmetrische Differenz $|p \oplus p'| = |p| + |p'| - 2|p \cap p'|$. Dies ergibt

$$|p'| \;=\; |p \oplus p'| - |p| + 2|p \cap p'| \overset{(4.3.2)}{\geq} |p| + 2|p \cap p'|.$$

$\square$

Wir wollen einen Algorithmus formulieren, der eine Folge von Matchings nach folgendem algorithmischen Gerüst aufbaut:

$M_0 = \emptyset$
für $i = 0, 1, \ldots$
    setze $M_{i+1} = M_i \oplus p_i$, wobei p_i kürzester zunehmender Pfad bzgl. M_i

Satz 4.3.19 zeigt, dass die Längen der p_i immer größer werden. Es gilt sogar

4.3.20 Lemma
Es gilt im obigen Algorithmus

(i) $|p_i| \leq |p_{i+1}|$

(ii) für alle $i \neq j$ mit $|p_i| = |p_j|$ sind p_i und p_j knotendisjunkt.

Beweis: nur zu (ii): Angenommen, $|p_i| = |p_j|$ für $i < j$ und p_i und p_j sind nicht knotendisjunkt. Dann existieren zwei Indizes k, ℓ mit $i \leq k < \ell \leq j$ so dass $|p_k| = |p_\ell|$, p_k und p_ℓ sind nicht knotendisjunkt, aber alle p_m sind knotendisjunkt mit p_k und p_ℓ, $k < m < \ell$. Der Pfad p_ℓ besitzt keine Kanten aus $p_{k+1}, \ldots, p_{\ell-1}$. Damit ist p_ℓ ein zunehmender Pfad nicht nur bzgl. M_ℓ

sondern auch bzgl. $M_{k+1} = M_k \oplus p_k$, also $|p_\ell| \geq |p_k| + |p_k \cap p_\ell|$ nach Satz 4.3.19. Wegen $|p_k| = |p_\ell|$ folgt daraus, dass p_k und p_ℓ keine gemeinsamen Kanten besitzen. Andererseits besitzen p_k und p_ℓ nach Annahme einen gemeinsamen Knoten v. Die Kante aus $M_k \oplus p_k$, mit welcher v inzident ist, gehört zu p_k und p_ℓ (denn p_ℓ ist zunehmender Pfad bzgl. $M_k \oplus p_k$). Dies ist ein Widerspruch. □

4.3.21 Satz
Sei s die Größe eines maximalen Matchings. Die Anzahl der verschiedenen Zahlen in der Folge

$$|p_0|,\ |p_1|, \ldots, |p_i|, \ldots, |p_{s-1}|$$

ist kleiner oder gleich $2\lfloor\sqrt{s}\rfloor + 1$.

Beweis: Nehme $r = \lfloor s - \sqrt{s}\rfloor < s$. Dann ist $|M_r| = r$ und nach Korollar 4.3.18 ist

$$\begin{aligned} |p_r| &\leq 2\left\lfloor \frac{r}{s-r} \right\rfloor + 1 = 2\left(\frac{s}{\lceil\sqrt{s}\rceil} - 1\right) + 1 \\ &\leq 2 \cdot \left(\frac{s}{\sqrt{s}} - 1\right) + 1 = 2\sqrt{s} - 1 \leq 2\lfloor\sqrt{s}\rfloor + 1. \end{aligned}$$

Damit: Die *verschiedenen* Werte von $|p_0|, \ldots, |p_r|$ sind höchstens alle ungeraden natürlichen Zahlen $\leq 2\lfloor\sqrt{s}\rfloor + 1$, also höchstens $\lfloor\sqrt{s}\rfloor + 1$ Stück. Anzahl der verschiedenen Werte in $|p_{r+1}|, \ldots, |p_{s-1}|$ sind höchstens $s-1-(r+1)+1 = s-r-1 \leq \lceil\sqrt{s}\rceil - 1 \leq \lfloor\sqrt{s}\rfloor$. Insgesamt also höchstens $2\lfloor\sqrt{s}\rfloor + 1$ verschiedene Werte. □

Auf der Grundlage von Lemma 4.3.20 können wir das Algorithmus-Gerüst wie folgt konkretisieren.

4.3.22 Algorithmus

$M = \emptyset$
while (es existiert ein zunehmender Pfad bzgl. M) **do**
{zentraler Schritt}
  sei $\ell(M)$ die Länge des kürzesten solchen Pfades
  finde eine maximale Menge $\{Q_1, \ldots, Q_t\}$ mit
    (i) jedes Q_i ist ein zunehmender Pfad bzgl. M der Länge $\ell(M)$
    (ii) die Q_i sind paarweise knotendisjunkt
  $M = M \oplus Q_1 \oplus \ldots \oplus Q_t$
end while

(Eine Menge heißt maximal bezüglich einer Eigenschaft, wenn keine echte Obermenge diese Eigenschaft besitzt.)
Beachte: Satz 4.3.21 zusammen mit Lemma 4.3.20 besagt, dass die while-Schleife aus zentralen Schritten höchstens $2\sqrt{s}+2$-mal durchlaufen wird (das letzte Mal, um festzustellen, dass kein zunehmender Pfad exisitiert). Wenn wir jeden zentralen Schritt mit Aufwand $\mathcal{O}(|E|+|V|)$ durchführen können, erhalten wir eine Gesamtkomplexität von $\mathcal{O}(\sqrt{|V|}(|E|+|V|))$, denn $s \leq |V|/2$.

Wir behandeln den zentralen Schritt jetzt für den Fall eines bipartiten Graphen $P=(V,E)$ mit $V=B\cup G$. Dazu sei $M \subseteq E$ das vorliegende Matching. Zu P definieren wir den gerichteten Graphen $D=(V,\bar{E})$, indem wir den Kanten Richtungen geben:

$$\begin{aligned}\bar{E} &= \{(u,v) \mid \{u,v\} \in M, u \in B, v \in G\} \\ &\quad \cup \{(v,u) \mid \{u,v\} \in E \setminus M, u \in B, v \in G\}.\end{aligned}$$

Jeder zunehmende Pfad bzgl. M ist damit ein gerichteter Pfad in D, der von einem freien 'girl' ($v \in G$) zu einem freien 'boy' ($u \in B$) führt. Wir konstruieren eine Teilgraphen $\hat{D}$ von D, dessen gerichtete Pfade von freien girls zu freien boys genau den kürzesten zunehmenden Pfaden bzgl. M (in D) entsprechen. Dazu konstruieren wir 'umgekehrte' Levelmengen $L_0, L_1, L_2, \ldots$.

$$\begin{aligned}L_0 &= \{u \in B,\ u \text{ ist frei }\} \quad \text{freie boys} \\ E_i &= \{(u,v) \mid (u,v) \in \bar{E}, v \in L_i, u \notin L_0 \cup \ldots \cup L_i\}, \\ L_{i+1} &= \{u \in V \mid \text{ ex. } v \in V \text{ mit } (u,v) \in E_i\}, \\ & \quad\; i = 0,1,2,\ldots\end{aligned}$$

Sei i^* die kleinste Zahl i mit $L_i \cap \{\text{ freie girls }\} \neq \emptyset$, und setze $\hat{D}=(\hat{V},\hat{E})$ mit

$$\begin{aligned}\hat{V} &= \bigcup_{i=0}^{i^*-1} L_i \cup (L_{i^*} \cap \{\text{ freie girls }\}), \\ \hat{E} &= \bigcup_{i=0}^{i^*-2} E_i \cup \{(u,v) \mid (u,v) \in E_{i^*-1}, u \text{ ist freies girl }\}.\end{aligned}$$

(s. Abbildung 4.3.)

4.3.23 Lemma
$\hat{D}$ hat folgende Eigenschaften:

- die Knoten der geraden Levels $L_0, L_2, \ldots$ sind alle boys, die der ungeraden $L_1, L_3, \ldots$ sind alle girls.
- $(u, v) \in \hat{E} \Rightarrow$ ex. i mit $v \in L_i, u \in L_{i+1}$.
- $\hat{D}$ enthält keine Zyklen.
- die kürzesten zunehmenden Pfade bzgl. M in E haben die Länge i^* und entsprechen eins zu eins den gerichteten Pfaden von einem freien girl zu einem freien boy in $\hat{D}$.

Beweis: klar auf Grund der Konstruktion von $\hat{D}$, s. auch Abbildung 4.3.

Wir ergänzen $\hat{V}$ um zwei Knoten q (Quelle) und s (Senke) und $\hat{E}$ um gerichtete Kanten von q zu allen freien girls $\in \hat{V}$ und von allen freien boys $\in \hat{V}$ zu s. Das Ergebnis nennen wir wieder $\hat{D} = (\hat{V}, \hat{E})$. Für den zentralen Schritt interessiert jetzt die Bestimmung einer maximalen Menge von (bis auf s und q) knotendisjunkten gerichteten Pfaden von q nach s. Dies erreicht man mittels einer Variante der klassischen Tiefensuche ähnlich zu Algorithmus 4.2.18.

4.3.24 Algorithmus

```
i = 1                                                    {Zähler für Pfade}
füge q in leeren Stack S ein
while  S ist nicht leer  do
  betrachte oberstes Element u von S
  if  es existiert ein nicht markierter Knoten v mit (u,v) in Ê then
    wähle einen solchen {falls v = s gibt's keine weiteren}
    if v = s then
      gebe S (ohne q) als gesuchten Pfad Q_i aus, i++
      füge q in nun wieder leeren Stack S ein
    else
      markiere v
      füge v in S ein
    end if
  else
    entferne u aus S
  end if
end while
```

Die Q_i sind knotendisjunkt, da bereits mit einem Q_i ausgegebene Knoten markiert sind. Der Algorithmus entfernt alle Knoten von $\hat{V}$ höchstens ein Mal aus S, er betrachtet insgesamt alle Kanten aus $\hat{E}$ höchstens einmal und

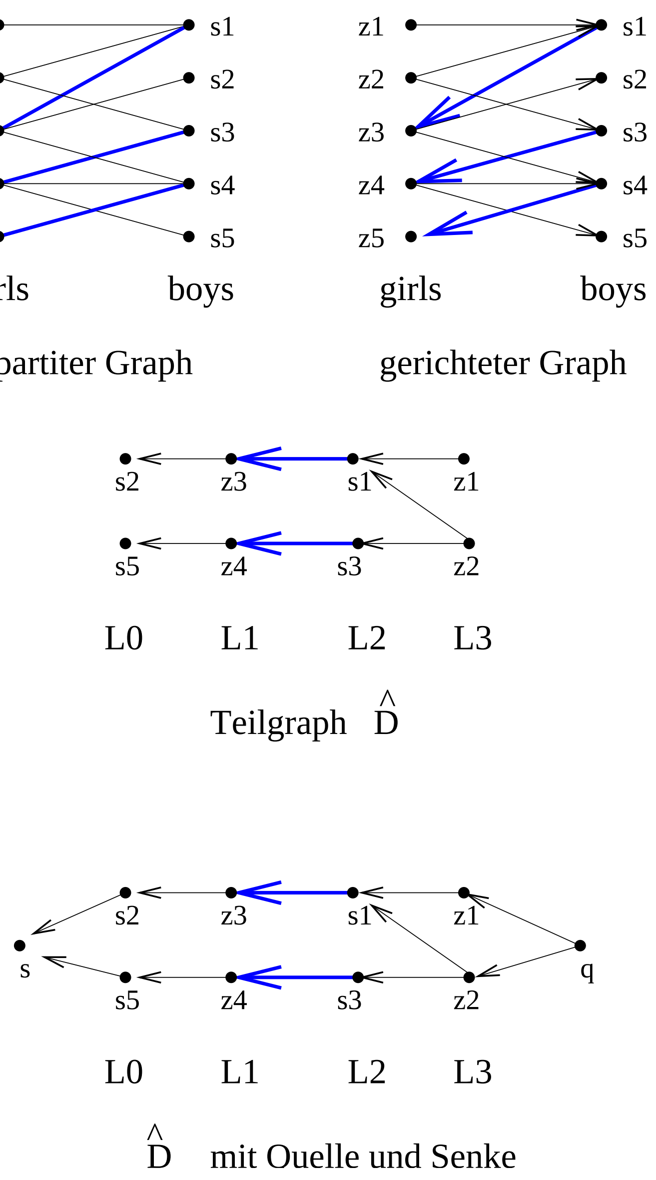


Abbildung 4.3: Matching im bipartiten Graphen und $\hat{D}$

er gibt in allen Q_i insgesamt höchstens alle Knoten aus, denn die Q_i sind knotendisjunkt. Deshalb ist der Aufwand $\mathcal{O}(|\hat{V}| + |\hat{E}|) = \mathcal{O}(|V| + |E|)$.

Angenommen, es gibt einen zu den ausgegebenen Q_i knotendisjunkten Pfad $(v_1, \ldots, v_m)$. Hierin ist v_1 markiert (denn der Algorithmus markiert alle zu q benachbarten Knoten). Ebenso ist v_m nicht markiert, denn sonst wäre v_m im nach Untersuchung der Kante (v_m, s) (die einzige, die von v_m ausgeht) ausgegebenen Pfad Q_i. Sei $k, 1 < k \leq m$ der kleinste Index, so dass v_k nicht markiert ist. Weil v_{k-1} markiert ist, wurde es in S eingefügt. Weil v_k nicht markiert ist, wurde die Kante (v_{k-1}, v_k) nie betrachtet, d.h. der Stack S wurde irgendwann einmal ausgegeben als Pfad Q_i, welcher v_{k-1} enthält. Damit ist Q nicht knotendisjunkt zu den Q_i, Widerspruch. Damit haben wir

4.3.25 Satz

(i) Algorithms 4.3.24 bestimmt eine maximale Menge von knotendisjunkten gerichteten Pfaden von q nach s mit Aufwand $\mathcal{O}(|\hat{V}| + |\hat{E}|) = \mathcal{O}(|V| + |E|)$.

(ii) Wendet man Algorithmus 4.3.22 mit Algorithmus 4.3.24 an auf den bipartiten Graphen $P(A)$ von $A \in \mathbb{R}^{n \times n}$, so beträgt sein Aufwand $\mathcal{O}(\sqrt{n}(n + nnz(A)))$.

Duff, Erisman und Reid berichten, dass der Algorithmus von Hopcroft und Karp wegen der aufwendigeren Implementierung dem von Hall in der Praxis doch unterlegen ist, obwohl die Worst-Case-Komplexität um die Ordnung $\sqrt{n}$ besser ist.

Abschnitt 4.4

B-reduzible Normalform

Erinnerung an Abschnitt 4.2: Reduzible Normalform (RNF)

$$PAP^T = \begin{bmatrix} A_{11} & & 0 \\ \vdots & \ddots & \\ A_{N1} & \dots & A_{NN} \end{bmatrix}, A_{ii} \in \mathbb{R}^{n_i \times n_i} \text{ irreduzibel}$$

In Abschnitt 4.2 haben wir (implizit) gezeigt:

4.4.1 Satz
Die RNF ist eindeutig bis auf

- Permutationen innerhalb jedes (Diagonal-) Blocks ($\widehat{=}$ starke Zusammenhangskomponente von $D(A)$),
- bestimmte Permutationen der Blöcke als Ganzes. ($\widehat{=}$ andere topologische Sortierung des Quotientengraphen).

Was gewinnt man, wenn man allgemeiner PAQ^T betrachtet, Q Permutationsmatrix?

4.4.2 Definition

(i) $A \in \mathbb{R}^{n \times n}$ heißt *beidseitig reduzibel* (*b-reduzibel*), falls Permutationsmatrizen P, Q existieren mit

$$PAQ^T = \begin{bmatrix} B_{11} & 0 \\ B_{21} & B_{22} \end{bmatrix}, B_{11}, B_{22} \text{ quadratisch und } n \geq 2$$

oder

$$(\ n = 1 \text{ und } A = 0).$$

Andernfalls heißt A *b-irreduzibel*.

(ii) Eine *b-reduzible Normalform* (BRNF) von A ist eine Darstellung

$$PAQ^T = \begin{bmatrix} B_{11} & & 0 \\ \vdots & \ddots & \\ B_{N1} & \dots & B_{NN} \end{bmatrix}, B_{ii} \in \mathbb{R}^{n_i \times n_i}, \tag{4.4.1}$$

mit

$$B_{ii} \begin{cases} \text{b-irreduzibel oder} \\ (\ n_i = 1 \text{ und } B_{ii} = 0). \end{cases}$$

Wir zeigen nun: Falls A eine Transversale besitzt, ist die BRNF „im Wesentlichen“ eindeutig und man erhält sie in zwei Schritten:

a) bestimme eine Transversale, d.h. P_1, so dass $(P_1A)_{ii} \neq 0, i = 1, \dots n$ (siehe Abschnitt 4.3)

b) bestimme P_2 so, dass $P_2(P_1A)P_2^T$ RNF von P_1A ist (siehe Abschnitt 4.2)

nehme $P = P_2P_1$, $Q = P_2$.

4.4.3 Lemma
Sei $B = PAQ^T$ eine BRNF von A, siehe (4.4.1). $\pi \mathrel{\widehat{=}} P$, $\sigma \mathrel{\widehat{=}} Q$, d.h. $b_{ij} = a_{\pi(i),\sigma(j)}$.

(i) AQ^T besitzt eine Transversale genau dann, wenn A eine solche besitzt.

(ii) A besitze eine Transversale, τ sei eine beliebige Transversale von AQ^T. Dann gilt

$$\begin{aligned}(4.4.2)\ \{\tau(m_i+1), \dots, \tau(m_{i+1})\} \quad = \quad & \{\pi(m_i+1), \dots, \pi(m_{i+1})\}, \\ & i = 1, \dots, N,\end{aligned}$$

wobei $m_i = \sum_{\ell=1}^{i-1} n_\ell$.

Beweis: Zu (i): s.u. (Lemma 4.4.5).
Zu (ii): Letzte Blockspalte von AQ^T hat Nichtnullen nur in den n_N Zeilen $\pi(m_{N-1}+1), \dots, \pi(m_N)$. Die Transversale τ permutiert diese Zeilen auf die Nummern $m_N + 1, \dots, m_{N+1}$. Dies beweist (4.4.2) für $i = N$. Streiche jetzt Blockzeile N und Blockspalte N und wiederhole das Argument induktiv. □

4.4.4 Folgerung
In der Situation von Lemma 4.4.3 gilt (T = Permutationsmatrix zu τ): $B_1 = TAQ^T$ ist ebenfalls BRNF (4.4.1) mit denselben Blockgrößen n_i.

4.4.5 Lemma
Sei $\Pi(A)$ die Menge aller Transversalen von A, Q sei Permutationsmatrix zur Permutation σ. Dann gilt

$$\Pi(AQ^T) = \Pi(A) \circ \sigma = \{\pi \circ \sigma, \pi \in \Pi(A)\}.$$

Beweis:

$$\begin{aligned}
\tilde{\pi} \in \Pi(AQ^T) &\iff (AQ^T)_{\tilde{\pi}(i),i} \neq 0, i = 1, \ldots, n \\
&\iff A_{\tilde{\pi}(i),\sigma(i)} \neq 0, i = 1, \ldots, n \\
&\iff A_{\tilde{\pi}(\sigma^{-1}(i)),i} \neq 0, i = 1, \ldots, n \\
&\iff \tilde{\pi} \circ \sigma^{-1} \in \Pi(A) \\
&\iff \tilde{\pi} = \pi \circ \sigma, \pi \in \Pi(A) \quad (\pi = \tilde{\pi} \circ \sigma^{-1}).
\end{aligned}$$

□

4.4.6 Folgerung
Sei $B = PAQ^T$ eine BRNF von A, τ sei Transversale von A, T zugehörige Permutationsmatrix. Aus

$$B = PAQ^T = (PT^T)(TA)Q^T \qquad (T^T = T^{-1})$$

ergibt sich nach Folgerung 4.4.4, dass $B' = T'(TA)Q^T$ mit $T' \mathrel{\widehat{=}}$ Transversale von $(TA)Q^T$ dieselbe Struktur (4.4.1) hat wie B, also auch BRNF ist. Es ist id Transversale von TA, nach Lemma 4.4.5 also $id \circ \sigma$ ($\sigma \mathrel{\widehat{=}} Q$) Transversale von $(TA)Q^T$. Insbesondere kann also $T' = Q$ gewählt werden und

$$Q(TA)Q^T = B''$$

besitzt die selbe Struktur (4.4.1) wie B', B.

4.4.7 Folgerung
Jede BRNF von A ergibt sich also als $Q(TA)Q^T$ für jede Transversale T. Die „Freiheiten“ bei den BRNF sind also neben der Wahl von T dieselben wie bei RNF (Satz 4.4.1). Insbesondere ist die BRNF in diesem Sinne „im Wesentlichen“ eindeutig. Sie wird berechnet durch die beschriebene 2-Stufen-Methode

1) bestimme Transversale $\tau \leftrightarrow T$,

2) bestimme RNF von TA.

4.4.8 Beispiel

$$A = \begin{bmatrix} 0 & \times & \times & \times & 0 \\ \times & 0 & 0 & \times & \times \\ 0 & 0 & \times & 0 & 0 \\ 0 & 0 & 0 & \times & \times \\ 0 & \times & 0 & \times & 0 \end{bmatrix}$$

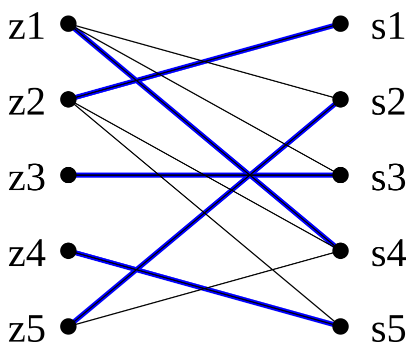
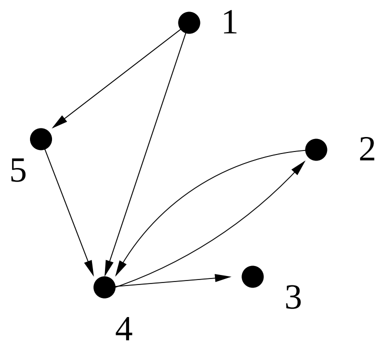


Abbildung 4.4: $P(A)$ für A aus Beispiel 4.4.8

Abbilding 4.4 zeigt $P(A)$. Man erkennt: Eine Transversale ist τ mit $\tau(1) = 4, \tau(2) = 1, \tau(3) = 3, \tau(4) = 5, \tau(5) = 2$. Damit ergibt sich

$$TA = \begin{bmatrix} \times & 0 & 0 & \times & \times \\ 0 & \times & 0 & \times & 0 \\ 0 & 0 & \times & 0 & 0 \\ 0 & \times & \times & \times & 0 \\ 0 & 0 & 0 & \times & \times \end{bmatrix}$$

$D(TA)$:



Digraph D(TA)

Starke Komponenten von $D(TA)$:

$$C_1 = \{1\},\ C_2 = \{2,4\},\ C_3 = \{3\},\ C_4 = \{5\}.$$

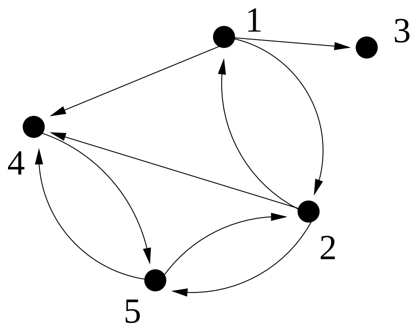
Umgekehrte topologische Sortierung des Quotientengraphen z.B.:

$$C_3,\ C_2,\ C_4,\ C_1$$

BRNF von A ist also

$$Q(TA)Q^T = \begin{bmatrix} \times & 0 & 0 & 0 & 0 \\ 0 & \times & \times & 0 & 0 \\ \times & \times & \times & 0 & 0 \\ 0 & 0 & \times & \times & 0 \\ 0 & 0 & \times & \times & \times \end{bmatrix}$$

Zum Vergleich: Ohne Transversale ist $D(A)$:



Digraph D(A)

mit den starken Zusammenhangskomponenten $C_1 = \{3\}$, $C_2 = \{1, 2, 4, 5\}$, bereits umgekehrt toplogisch sortiert. Die RNF von A ist also

$$PAP^T = \begin{bmatrix} \times & 0 & 0 & 0 & 0 \\ 0 & \times & 0 & \times & \times \\ 0 & 0 & \times & \times & 0 \\ \times & 0 & \times & \times & 0 \\ 0 & 0 & 0 & \times & \times \end{bmatrix}$$

Abschnitt 4.5

Rückgriff auf symmetrische Strukturen

Gegeben: $A \in \mathbb{R}^{n\times n}$, nicht symmetrisch, beidseitig irreduzibel. Vorgehen zur Faktorisierung (wird in der Praxis häufig verwendet):

a) bilde die symmetrische Struktur $\mathit{struct}(A + A^T)$,

b) bestimme mit einer der Methoden aus Kapitel 3 eine Permutation P (für $A + A^T$),

c) $A \leftarrow PAP^T$,

d) faktorisiere A als $A = LDU$, L in die L-Datenstruktur von $P(A + A^T)P^T$, U in (L-Datenstruktur)^T von $P(A + A^T)P^T$,

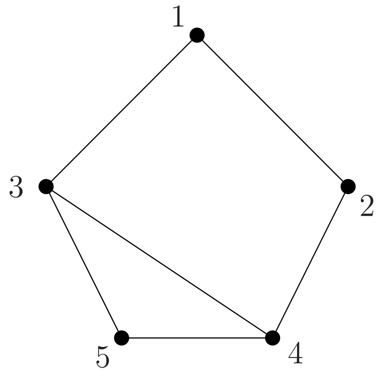
e) schätze ρ nach oben ab (Abschnitt 4.1).

In diesem Fall hat man also wieder eine symbolische und numerische Phase. Falls $\mathit{struct}(A)$ unsymmetrisch ist, sind in L und U einige „nicht-zufällige“ Nullen explizit gespeichert.

4.5.1 Beispiel

$$A = \begin{bmatrix} \times & 0 & \times & 0 & 0 \\ \times & \times & 0 & \times & 0 \\ 0 & 0 & \times & \times & \times \\ 0 & \times & 0 & \times & \times \\ 0 & 0 & \times & 0 & \times \end{bmatrix} \qquad A + A^T = \begin{bmatrix} \times & \otimes & \times & 0 & 0 \\ \times & \times & 0 & \times & 0 \\ \otimes & 0 & \times & \times & \times \\ 0 & \times & \otimes & \times & \times \\ 0 & 0 & \times & \otimes & \times \end{bmatrix}$$

$G(A + A^T)$:



MD-Anordnung für $A + A^T$: $G(A + A^T)$ ist z.B. $1, 2, 3, 4, 5 \mathrel{\widehat{=}} id$, d.h. $A + A^T$ ist bereits in MD-Anordnung. Bei Gauß-Elimination ergibt sich

$$\begin{bmatrix} \times & \otimes & \times & 0 & 0 \\ \times & \times & 0 & \times & 0 \\ \otimes & 0 & \times & \times & \times \\ 0 & \times & \otimes & \times & \times \\ 0 & 0 & \times & \otimes & \times \end{bmatrix} \rightsquigarrow \begin{bmatrix} \times & \otimes & \times & 0 & 0 \\ \times & \times & \times & \times & 0 \\ \otimes & \otimes & \times & \times & \times \\ 0 & \times & \otimes & \times & \times \\ 0 & 0 & \times & \otimes & \times \end{bmatrix}$$

$$\rightsquigarrow \begin{bmatrix} \times & \otimes & \times & 0 & 0 \\ \times & \times & \times & \times & 0 \\ \otimes & \otimes & \times & \times & \times \\ 0 & \times & \times & \times & \times \\ 0 & 0 & \times & \otimes & \times \end{bmatrix} \rightsquigarrow \ldots \begin{bmatrix} \times & \otimes & \times & 0 & 0 \\ \times & \times & \times & \times & 0 \\ \otimes & \otimes & \times & \times & \times \\ 0 & \times & \times & \times & \times \\ 0 & 0 & \times & \times & \times \end{bmatrix}$$

$$\rightsquigarrow L = \begin{pmatrix} \times & & & & \\ \times & \times & & & \\ \otimes & \otimes & \times & & \\ 0 & \times & \times & \times & \\ 0 & 0 & \times & \times & \times \end{pmatrix} \qquad U = \begin{pmatrix} \times & \otimes & \times & 0 & 0 \\ & \times & \times & \times & 0 \\ & & \times & \times & \times \\ & & & \times & \times \\ & & & & \times \end{pmatrix}$$

Die $\otimes$ markieren Positionen, die in der Datenstruktur vorhanden sind, aber mit Wert 0 belegt sind.
$nnz(L) = 12$; bei Berücksichtigung der Nullen $nnz(L) = 10$,
$nnz(U) = 12$; bei Berücksichtigung der Nullen $nnz(U) = 11$.

Die beschriebene Vorgehensweise ignoriert Probleme, die wegen zu kleiner (oder gar verschwindender) Pivotelemente auftreten können. Man erkennt solche Probleme aber, wenn die Schranke für den Wachstumsfaktor ρ zu groß wird.

Eine Heurstik zur Vermeidung kleiner Pivots ist, anfänglich große Diagonalelemente zu erreichen. Wir beschreiben eine kürzlich vorgestellte solche Heuristik im Folgenden[1].

4.5.2 Definition
Sei $A \in \mathbb{R}^{n\times n}$. Dann ist

$$\mathrm{diagmin}(A) = \min_{i=1}^{n} |a_{ii}|.$$

4.5.3 Definition
Eine Transversale τ für $A \in \mathbb{R}^{n\times n}$, A nicht strukturell singulär, heißt *Flaschenhals-Transversale*, falls gilt

$$\mathrm{diagmin}(TA) = \max_{\pi \text{Transversale}} \mathrm{diagmin}(PA),$$

wobei T, P Permutationsmatrix zu τ, π. Wir schreiben

$$\mathrm{diagmin}(TA) = \mathrm{bn}(A).$$

Eine Flaschenhalstransversale τ ist also charakterisiert durch

$$\min_{i=1}^{n} |a_{\tau(i),i}| = \max_{\pi \text{Transversale}} \min_{i=1}^{n} |a_{\pi(i),i)}|.$$

4.5.4 Lemma
Ist τ Flaschenhalstransversale für $A \in \mathbb{R}^{n\times n}$, T zugeh. Permutationsmatrix, so gilt für jede Wahl von Permutationsmatrizen P, Q

$$\mathrm{diagmin}(TA) \geq \mathrm{diagmin}(PAQ^T).$$

Folgerung: Eine Flaschenhalstransversale bringt „möglichst" große Einträge auf die Diagonale.
Wir kümmern uns nun um die algorithmische Bestimmung von τ. Dazu

4.5.5 Definition
Sei $A \in \mathbb{R}^{n\times n}$. Zu $\epsilon \geq 0$ ist die Matrix $A^{\geq\epsilon}$ definiert als

$$(A^{\geq\epsilon})_{ij} = \begin{cases} a_{ij} & \text{falls } |a_{ij}| \geq \epsilon \\ 0 & \text{sonst} \end{cases}$$

[1] Duff, I., Koster, J.: The design and use of algorithms for permuting large entries to the diagonal of sparse matrices, SIAM J. Matrix Anal. Appl. **20**, 889-901 (1999)

Es ist also $A = A^{\geq 0}$.
Folge: Ist $\epsilon_0 = \mathrm{bn}(A)$, so besitzt $A^{\geq \epsilon}$ Transversalen für alle $\epsilon \leq \epsilon_0$, und keine Transversale für $\epsilon > \epsilon_0$.
In Zukunft sagen wir kurz „M ist ein Matching für A", wenn es eines für den bipartiten Graphen $P(A)$ (siehe Definition 4.3.12) ist. M muss nicht unbedingt die Größe n haben.
Beachte: Für $\epsilon > \epsilon'$ ist $P(A^{\geq \epsilon})$ ein Teilgraph von $P(A^{\geq \epsilon'})$ (mit evtl. weniger Kanten). Jedes Matching für $P(A^{\geq \epsilon})$ ist auch eines für $P(A^{\geq \epsilon'}$, aber nicht umgekehrt.

4.5.6 Definition
Sei M ein Matching für A und $\epsilon \geq 0$. Dann ist

$$M^{\geq \epsilon} = \{(i,j) \in M \mid |a_{i,j}| \geq \epsilon\}$$

Es ist $M^{\geq \epsilon} \subseteq M$ ein Matching von A und $A^{\geq \epsilon}$.
Wir gehen nun davon aus, dass wir ein Verfahren haben, das ein Matching M für A erweitert zu einem maximalen Matching $M' = MM(A, M)$. (Die Verfahren von Hall und von Hopcroft und Karp können beide entsprechend modifiziert werden.)

Idee für Algorithmus: Verkleinere durch Bisektion sukzessive ein Intervall $[\underline{\epsilon}, \overline{\epsilon}]$, so dass $A_{\underline{\epsilon}}$ kein, $A_{\overline{\epsilon}}$ dagegen ein Matching der Größe n besitzt.

4.5.7 Algorithmus
(Flaschenhals-Matching)

setze $\underline{\epsilon} = 0$, $\overline{\epsilon} = \infty$
$M = MM(A, \emptyset)$
$T' = M$
while es ex. i, j mit $\underline{\epsilon} < |a_{ij}| < \overline{\epsilon}$ **do**
  wähle ein solches (i, j), setze $\epsilon = |a_{ij}|$ {Details später}
  $T = MM(A^{\geq \epsilon}, T'_\epsilon)$
  if $|T| = |M|$ **then**
    $T' = T$, $\underline{\epsilon} = \epsilon$
  else
    $\overline{\epsilon} = \epsilon$
  end if
end while

Wenn A eine Transversale besitzt, ist $|M| = n$. Der Algorithmus ist so formuliert, dass er auch im Falle einer strukturell singulären Matrix ein maximales Flaschenhalsmatching bestimmt.

Korrektheit des Algorithmus: Scheifeninvariante für die While-Schleife ist:

> T' ist ein maximales Matching für $A^{\geq \underline{\epsilon}}$, $A^{\geq \overline{\epsilon}}$ besitzt kein maximales Matching.

Die While-Schleife bricht ab, denn die Zahl der möglichen Kandidaten i, j verringert sich pro Durchlauf um mindestens eins. Bei Abbruch ist $\underline{\epsilon} \leq \mathrm{bn}(A) < \overline{\epsilon}$ und T' ist ein Flaschenhals-Matching.
Zur Wahl von (i, j) mit

$$\underline{\epsilon} < \epsilon = |a_{ij}| < \overline{\epsilon} \tag{4.5.1}$$

in der while-Schleife:

a) Versuche, $\epsilon \approx (\underline{\epsilon} + \overline{\epsilon})/2$ zu erreichen.
Dann wird in jedem Schritt das Intervall $[\underline{\epsilon}, \overline{\epsilon}]$ ungefähr halbiert.

b) Wähle ϵ als den *Median* aller $|a_{ij}|$ mit $\underline{\epsilon} \leq |a_{ij}| < \overline{\epsilon}$. Dann wird die Zahl der zu betrachtenden (i, j) in jedem Schritt halbiert. Das ist i. A. schneller.

c) Durchlaufe Spalten (Zeilen) von $A^{\geq \underline{\epsilon}}$ und sammle so k (z.B. $k = 10$) Positionen (i_ℓ, j_ℓ) mit (4.5.1). Nehme den Median dieser k Positionen. (In Praxis bevorzugte Variante.)

Zur Implementierung:

- Es ist wichtig, $A^{\geq \underline{\epsilon}}$ und $A^{\geq \overline{\epsilon}}$ einfach aufdatieren zu können, wenn sich $\underline{\epsilon}, \overline{\epsilon}$ ändern. Für a) und b) muss man dazu A (implizit) sortieren (ein Feld von Pointern realisiert die Vertauschungen).
Aufwand aber $\mathcal{O}(nnz(A) \log(nnz(A)))$.

- Für c) genügt es, wenn jede Spalte (Zeile) von A sortiert wird, Aufwand $\mathcal{O}(n)$ (wenn pro Zeile $\mathcal{O}(1)$ Einträge).

- Bei Verwendung des Algorithmus von Hall sollte man im Look-ahead Schritt immer möglichst große Einträge nehmen, wenn man da eine Wahl hat. So werden im Matching große Zahlen gegnüber kleineren bevorzugt.

- In der Praxis wird man die while-Schleife abbrechen, sobald $\overline{\epsilon} - \underline{\epsilon}$ klein genug ist.

- Im Falle $|T| = |M|$ kann man $\underline{\epsilon}$ durch $\underline{\epsilon} = \min_{(i,j) \in T} |a_{i,j}|$ vergößern.

Abschnitt 4.6

Markowitz-Strategie

Ausgangspunkt sei wie in Abschnitt 4.5. Bei der Markowitz-Strategie wird Gauß-Elimination mit Spalten- **und** Zeilen-Pivotsuche verwendet.

4.6.1 Algorithmus
(Gauß-Elimination mit beidseitiger Pivotsuche, explizite Vertauschungen.)

for $k = 1$ to $n-1$ **do**
  bestimme Pivotelement $a_{st}^{(k)}$ in $A^{(k)}(k:n, k:n)$ {Pivotwahl}
  vertausche Zeilen k und s sowie die Spalten k und t in $A^{(k)}$, ergibt $\tilde{A}^{(k)}$.
  for $i = k+1$ to n **do**
    $\ell_{ik} = \tilde{a}_{ik}^{(k)} / \tilde{a}_{kk}^{(k)}$
  end for
  for $i = k+1$ to n **do**
    for for $j = k+1$ to n **do**
      $a_{ij}^{(k+1)} = \tilde{a}_{ij}^{(k)} - \ell_{ik}\tilde{a}_{kj}^{(k)}$
    end for
  end for
end for

Bestimmt wird also $PAQ^T = LU$, P, Q Permutationsmatrizen.
Bemerkung: Wählt man $|a_{st}^{(k)}| = \max\limits_{i,j=k}^{n} |a_{ij}^{(k)}|$ (vollständige Pivotsuche, ist $\mathcal{O}(n^2)$-Falle), so gilt für den Wachstumsfaktor:

$$\rho \leq \sqrt{n \cdot 2^1 \cdot 3^{\frac{1}{2}} \cdot 4^{\frac{1}{3}} \dots n^{\frac{1}{n-1}}} \cdot \max_{i,j=1}^{n} |a_{ij}|.$$

(Wilkinson, 1961)

4.6.2 Definition

$$\begin{aligned} r_i^{(k)} &= |struct(A^{(k)}(i, k:n))| \quad \text{(Anzahl der Nichtnullen in Zeile } i\text{)} \\ c_j^{(k)} &= |struct(A^{(k)}(k:n, j))| \quad \text{(Anzahl der Nichtnullen in Spalte } j\text{)} \end{aligned}$$

$$m_{ij}^{(k)} \quad = \quad (r_i^{(k)} - 1) \cdot (c_j^{(k)} - 1) \quad \text{„Markowitz-Zahl".}$$

4.6.3 Definition
(Markowitz-Strategie)

a) ohne Rücksicht auf Stabilität: wähle Pivot-Element s, t so, dass

$$m_{st}^{(k)} = \min\{m_{ij}^{(k)} : i, j = k, \ldots, n \text{ und } a_{ij}^{(k)} \neq 0\}.$$

b) mit Schwellenpivotsuche (Parameter u): wähle Pivot s, t so, dass

$$(4.6.1)\quad m_{st}^{(k)} = \min\left\{m_{ij}^{(k)} : i, j = k, \ldots, n \text{ und } |a_{ij}^{(k)}| \geq u \cdot \max_{\ell=k}^{n} |a_{\ell j}^{(k)}|\right\}.$$

Begründung: Die Positionen der Elemente von $A^{(k)}$, welche im Schritt k neu bestimmt werden, sind

$$(\mathit{struct}(A_{\cdot t}(k:n)) \setminus \{t\}) \times (\mathit{struct}(A_{s\cdot}(k:n)) \setminus \{s\}).$$

Deren Zahl ist $m_{st}^{(k)}$; sie wird bei Strategie a) minimiert. Bei b) wird sie minimiert unter der Nebenbedingung

$$|a_{st}^{(k)}| \geq u \cdot \max_{\ell=k}^{n} |a_{\ell t}^{(k)}|$$

(numerische Stabilität).
Markowitz ist wie MD also eine Greedy-Strategie zur Minimierung des Rechenaufwandes.

4.6.4 Beispiel
Permutationen mit Markowitz-Strategie:

$$A = \begin{bmatrix} \times & 0 & \times & 0 & 0 \\ \times & \times & 0 & \times & 0 \\ 0 & 0 & \times & \times & \times \\ 0 & \otimes & 0 & 0 & 0 \\ 0 & 0 & \times & 0 & \times \end{bmatrix} \rightsquigarrow \begin{bmatrix} \times & 0 & 0 & 0 & 0 \\ \times & \otimes & 0 & \times & 0 \\ 0 & 0 & \times & \times & \times \\ 0 & \times & \times & 0 & 0 \\ 0 & 0 & \times & 0 & \times \end{bmatrix}$$

$$\rightsquigarrow \begin{bmatrix} \times & 0 & 0 & 0 & 0 \\ \times & \times & 0 & \times & 0 \\ 0 & 0 & \times & \times & \times \\ 0 & \times & \times & \times & 0 \\ 0 & 0 & \times & 0 & \otimes \end{bmatrix} \rightsquigarrow \begin{bmatrix} \times & 0 & 0 & 0 & 0 \\ \times & \times & 0 & \times & 0 \\ 0 & 0 & \times & 0 & \times \\ 0 & \times & 0 & \times & \times \\ 0 & 0 & \times & \times & \times \end{bmatrix}$$

$PAQ^T = LU$:

$$L = \begin{pmatrix} \times & & & & \\ \times & \times & & & \\ 0 & 0 & \times & & \\ 0 & \times & 0 & \times & \\ 0 & 0 & \times & \times & \times \end{pmatrix} \qquad U = \begin{pmatrix} \times & 0 & 0 & 0 & 0 \\ & \times & 0 & \times & 0 \\ & & \times & 0 & \times \\ & & & \times & \times \\ & & & & \times \end{pmatrix}$$

$$nnz(L) = 9,\ nnz(U) = 8.$$

(Vergleiche mit B aus Abschnitt 4.5.)

4.6.5 Satz
A sei symmetrisch. Dann ist die Markowitz-Strategie (ohne Rücksicht auf Stabilität) identisch mit der Minimum-Degree-Methode.

Beweis: Übung.

Für die Implementierung gilt:

- es sind $struct(L)$ und $struct(U)$ zu ermitteln, in Schritt k muss

  $$struct(A^{(k)}(k:n, k:n))$$

  bekannt sein.

- In der Variante „ohne Rücksicht auf die Stabiltät“ kann man eine symbolische Phase vorschalten. Dies geht „im Prinzip“ (bis auf die Nicht-Symmetrie) wie im symmetrischen Fall (Abschnitt 2.3).

- Bei Schwellenpivotsuche wird die Struktur während der numerischen Phase mit bestimmt. Numerische und symbolische Phase sind nicht mehr voneinander zu trennen,

- das Auffinden von s, t darf nicht zur $\mathcal{O}(n^2)$-Falle werden.

Wir behandeln zuerst den letzten Punkt. Wir modifizieren zuerst die Schwellenpivotbedingung zu

(4.6.2) $$m_{\overline{st}}^{(k)} \quad \min\Big\{ m_{ij}^{(k)} : i,j = k,\dots,n \text{ und } \Big(|a_{ij}^{(k)}| \geq u \cdot \max_{\ell=k}^{n} |a_{\ell j}^{(k)}| \text{ oder } |a_{ij}^{(k)}| \geq u \cdot \max_{\ell=k}^{n} |a_{i\ell}^{(k)}| \Big) \Big\}.$$

(Spalten- oder Zeilenpivotsuche!)

Damit werden also entweder die Elemente der Spalte k von L oder die der Zeile k von U alle $\leq 1/u$.
Überlegung: Angenommen, es existiert ein $c > 0$ mit $nnz(A^{(k)}(k:n, k:n)) \geq c \cdot (n-k)$ (ist wohl fast immer der Fall), so ist die Bestimmung der $m_{ij}^{(k)}$ ein Aufwand $\Omega(c(n-k))$; über alle Schritte $(k = 1, \ldots, n)$ also $\Omega(cn^2)$ (!)

Man sollte also nicht alle m_{ij}^k wirklich berechnen.
Wie bei Minimum-Degree verwenden wir für die $r_i^{(k)}$ und die $c_j^{(k)}$ eine Listenverwaltung mit Hilfe von Feldern (siehe Abschnitt 3.3), so dass Einfügen und Löschen mit Aufwand $\mathcal{O}(1)$ möglich wird. Benötigte Listen:

$$\mathcal{C}_\mu\text{: (Liste } \mathcal{C}_\mu \text{ enthält Knoten } j \text{ mit } c_j^{(k)} = \mu),$$
$$\mathcal{R}_\nu\text{: (Liste } \mathcal{R}_\nu \text{ enthält Knoten } i \text{ mit } r_i^{(k)} = \nu).$$

4.6.6 Beispiel

$$A = \begin{bmatrix} \times & 0 & \times & 0 & 0 \\ \times & \times & 0 & \times & 0 \\ 0 & 0 & \times & \times & \times \\ 0 & \otimes & 0 & 0 & 0 \\ 0 & 0 & \times & 0 & \times \end{bmatrix}$$

Alle Listen $\mathcal{R}_\nu$ mit drei Feldern *rlheaders*, *rlvorg*, *rlnachf*, alle Listen $\mathcal{C}_\mu$ mit drei Felder *clheaders*, *clvorg*, *clnachf*:

ν / i	1	2	3	4	5
rlheaders	4	5	3	–	–
rlvorg	0	0	2	0	1
rlnachf	5	3	0	0	0

μ / i	1	2	3	4	5
clheaders	–	5	3	–	–
clvorg	0	1	0	2	5
clnachf	2	4	0	4	0

Wir nennen diese sechs Felder zusammen die *Markowitz-Liste*.
Im Gegensatz zu MD verwalten wir aber *keine* Liste der m_{ij}^k.

Vorgehen zur Bestimmung von s, t nun schrittweise:
In Schritt ℓ seien alle Listen $\mathcal{C}_j$, $\mathcal{R}_i$, $i, j = 1, \ldots, \ell$ bereits durchsucht, $(\tilde{s}, \tilde{t})$ sei bisher ermitteltes Indexpaar, welches (4.6.2) erfüllt für bisher untersuchte i, j.

1.Fall: $m_{\tilde{s}\tilde{t}} \leq (\ell - 1)^2$
dann stopp; $(\tilde{s}, \tilde{t})$ ist gesuchtes Indexpaar, denn für alle nicht untersuchten Spalten und Zeilen ist $c_j \geq \ell$, $r_i \geq \ell$, also $m_{ij} \geq (\ell - 1)^2$.

2.Fall: $m_{\tilde{s}\tilde{t}} > (\ell - 1)^2$
durchlaufe $\mathcal{R}_{\ell+1}$ und $\mathcal{C}_{\ell+1}$.

Algorithmus dazu (ohne oberen Index ^{k}):

4.6.7 Algorithmus
(Bestimmung des Pivots bei Markowitz-Strategie)

$\ell = 1,\ m = n^2$
while $m \geq (\ell-1)^2$ **do**
{Zeilen mit Länge ℓ}
  for alle $i \in \mathcal{R}_\ell$ **do**
    compute $th = \max_{j \in struct(A_{i,\cdot})} |a_{ij}|$ {für Schwellenpivot}
    for alle $j \in struct(A_{i,\cdot})$ **do**
      berechne $m_{ij} = (\ell - 1) \cdot (c_j - 1)$
      if $m_{ij} < m$ und $|a_{ij}| \geq u \cdot th$ **then**
        $m = m_{ij},\ s = i; t = j$
      end if
    end for
  end for
{Spalten mit Länge ℓ}
  for alle $i \in \mathcal{C}_\ell$ **do**
    compute $th = \max_{i \in struct(A_{\cdot,j})} |a_{ij}|$ {für Schwellenpivot}
    for alle $i \in struct(A_{\cdot,j})$ **do**
      berechne $m_{ij} = (r_i - 1) \cdot (\ell - 1)$
      if $m_{ij} < m$ und $|a_{ij}| \geq u \cdot th$ **then**
        $m = m_{ij},\ s = i; t = j$
      end if
    end for
  end for
  $\ell = \ell + 1$
end while

Bemerkungen:

- Auf die zweite äußere **for**-Schleife kann man verzichten, falls $m < \ell(\ell-1)$, denn dann ist die Markowitz-Position bereits gefunden.
- c_j und r_i sollten für jedes j, i mit Aufwand $\mathcal{O}(1)$ bestimmbar sein $\rightsquigarrow$ Ergänzung unserer üblichen Datenstrukturen notwendig. Man kann z.B. je ein Feld verwenden, welches nach jedem Schritt aufdatiert wird.
- Aufwand für einen Lauf durch die **while**-Schleife:

$$\sum_{i \in \mathcal{R}_\ell} \mathcal{O}(r_i) + \sum_{j \in \mathcal{C}_\ell} \mathcal{O}(c_j) \ = \ (|\mathcal{R}_\ell| + |\mathcal{C}_\ell|) \cdot \ell.$$

- Der Algorithmus kann auch dann im Prinzip immer noch zur $\mathcal{O}(n^2)$-Falle werden, in der Praxis aber nicht.
- Ist $\mathcal{R}_1$ oder $\mathcal{C}_1$ nicht leer, so benötigt der Algroithmus nur einen Durchlauf durch die äußere **for**-Schleife (und es ist $m = 0$).

Jetzt kümmern wir uns um eine geeignete LDU-Datenstruktur für die Markowitz-Strategie.

4.6.8 Definition
Die DCEL-Datenstruktur („doubly connected edge list") für eine unsymmetrische Matrix A ist die DCEL für $D(A)$, wobei als Information für jede Kante (i, j) gespeichert sind: $a_{i,j}$, i und j. Insbesondere gilt:

- jede Kante (i, j) kommt in genau zwei doppelt verketteten Listen vor, nämlich der Liste $\mathcal{A}_i$ der Kanten mit Anfangsknoten i und der Liste $\mathcal{E}_j$ der Kanten mit Endpunkt j.
- Jede Kante (i, j) mit der Information a_{ij}, i und j ist dabei nur einmal abgespeichert.

In Info II im Grundstudium wurde eine Implementierung der DCEL vorgestellt. Bei der in dieser Vorlesung bevorzugten Realisierung über Felder benötigt man die Felder $va\ell, cn, rn$ für die Kanteninformation a_{ij}, i und j sowie die Felder *aheaders*, *avorg*, *anachf* und *eheaders*, *evorg*, *enachf* für die Listen.

4.6.9 Beispiel

$$A = \begin{bmatrix} 1.1 & 0 & 2.2 & 0 & 0 \\ 3.3 & 4.4 & 0 & 5.5 & 0 \\ 0 & 0 & 6.6 & 7.7 & 8.8 \\ 0 & 9.9 & 0 & 0 & 0 \\ 0 & 0 & -1.1 & 0 & -2.2 \end{bmatrix}$$

Eine DCEL-Darstellung ist

i,j,k	1	2	3	4	5	6	7	8	9	10	11
val	1.1	2.2	3.3	4.4	5.5	6.6	7.7	8.8	9.9	-1.1	-2.2
rn	1	1	2	2	2	3	3	3	4	5	5
cn	1	3	1	2	4	3	4	5	2	3	5
$aheaders$	1	3	6	9	10						
$avorg$	0	1	0	3	4	0	6	7	0	0	10
$anachf$	2	0	4	5	0	7	8	0	0	11	0
$eheaders$	1	4	2	5	8						
$evorg$	0	0	1	0	0	2	5	0	4	6	8
$enachf$	3	6	0	9	7	10	0	11	0	0	0

Mit der DCEL-Datenstruktur müssen wir Operationen vornehmen, deren Aufwand wir im folgenden zusammenfassen.

4.6.10 Lemma
Mit der Bezeichnung r_i (c_j) für die Länge der i-ten Zeile (j-ten Spalte) gilt für die DCEL-Darstellung

a) das Einfügen eines neuen Eintrags a_{ij} hat Aufwand $\mathcal{O}(1)$

b) das Auslesen oder Löschen der i-ten Zeile (j-ten Spalte) hat Aufwand r_i (c_j).

c) das (explizite) Vertauschen von Zeilen i und k besitzt den Aufwand

$$\mathcal{O}(\sum_{j \in struct(A_{i,\cdot}) \cup struct(A_{k,\cdot})} c_j \ ;$$

analog für Spalten

d) die Bestimmung des betragsgrößten Elements von Zeile i (Spalte j) besitzt den Aufwand $\mathcal{O}(r_i)$ ($\mathcal{O}(c_j)$).

e) ein Schritt der Gauß-Elimination (nach Vertauschungen wegen Pivotwahl), d.h. die SAXPYs mit Zeile k und Zeile j, $j \in struct(A_{\cdot,k})$, besitzt den Aufwand

$$\mathcal{O}(\sum_{j \in struct(A_{\cdot,k}), j \neq k} r_j + r_k) = \mathcal{O}((c_k - 1) r_k) + \mathcal{O}(\sum_{j \in struct(A_{\cdot,k}), j \neq k} r_j + r_k).$$

Das ist mehr als der reine Rechenaufwand $\mathcal{O}(r_k(c_k - 1))$!)

f) die Aufdatierung von r_i und c_j in der Markowitz-Liste erfordert Aufwand $\mathcal{O}(r_k c_k)$.

Beweis: Teil a) und b) sind im Wesentlichen aus Info II bekannt. Teil c) wurde als Algorithmus 1.2.10 für das erweiterte Listenformat angegeben, für DCEL geht er genau so. Teil d) ist klar (man durchlaufe die Zeile/Spalte). Teil e) ergibt sich, wenn man für die Aufdatierung einer jeden Zeile j eine Variante des SAXPY-Algorithmus 1.1.6 durchführt: scatter k-te Zeile von A in (vollen) Hilfsvektor w, durchlaufe Zeile j und datiere sie auf, verlängere Zeile j um die fill-in Beiträge, setze dabei w auf Null zurück; Aufwand $\mathcal{O}(r_k + r_j)$. Wenn man bei der Berechnung der Fill-ins gleichzeitig r_j und die betroffenen c_j aufdatiert, jeweils mit Aufwand $\mathcal{O}(1)$ und jeweils mit Aufwand $\mathcal{O}(1)$ die Markowitz-Liste abändert, ergibt sich für f) ein Aufwand von r_k pro aufdatierte Zeile, also pro Schritt $\mathcal{O}(r_k c_k)$.? □

Bemerkungen:

- Die doppelte Verkettung der Listen ist für unsere Zwecke eigentlich nicht nötig. Es genügt eine „SCEL“, d.h. *avorg*aund *evorg*können entfallen.

- Im k-ten Schritt der Gauß-Elimination benötigen wir die Matrix $A^{(k)}$ mit entsprechend verkürzten Zeilen und Spalten. Das haben wir in unserer Notation nicht berücksichtigt. Praktisch wird beim Abspeichern des Resultats des SAXPYs so vorgegangen: Beiträge zur k-ten Spalte werden mit dem entsprechenden Wert für L überschrieben und aus der *Zeilen*liste *entfernt*, nicht aber aus der Spaltenliste. Am Ende enthalten so die Spaltenlisten also L, die Zeilenlisten enthalten U, die Diagonale D wird extra gespeichert.

KAPITEL 5

Ein Blick auf die aktuelle Forschung

Dieses Kapitel soll ein paar Aspekte aktueller Forschung ansprechen.

Abschnitt 5.1

Dichte Diagonalblöcke

Jetzt: Kopplung von graphentheoretischen Algorithmen mit *iterativen* Verfahren für $Ax = b$.

Wir betrachten Iterationsverfahren nur ganz grob: Bestimmung von Iterierten

$$x^0, x^1, \ldots, x^k, \ldots$$

wobei x^{k+1} „einfach“ aus x^k (und evtl. weiteren alten Iterierten) und A und b zu berechnen ist.

5.1.1 Beispiel
Jacobi-Iteration:

$$x_i^{k+1} = \frac{1}{a_{ii}} \cdot \left(b_i - \sum_{\ell=1, \ell \neq i}^{n} a_{i\ell} x_\ell^k \right), \; i = 1, \ldots, n.$$

Mit $D = \operatorname{diag}(A)$ und $B = \operatorname{diag}(A) - A$ schreibt man das als

$$x^{k+1} = D^{-1}(Bx^k + b).$$

Block-Jacobi-Iteration: Man nehme D als einen Block-Diagonalteil von A, $B = D - A$ und rechne wieder

$$x^{k+1} = D^{-1}(Bx^k + b).$$

Wir gehen nicht auf Konvergenztheorie ein. Bei einer dünn besetzten Matrix ist man daran interessiert, im Block-Jacobi-Verfahren möglichst volle Diagonalblöcke zu bekommen. Erwartet wird dann eine schnellere Konvergenz.
Wir behandeln jetzt ein graphentheoretisches Verfahren, das dichte Diagonalblöcke erzeugt per Transformation $A \to PAP^T$.

Idee: Bilde Diagonalblöcke sukzessive durch Aufnahme geeigneter adjazenter Knoten. Achte auf Komplexität $\mathcal{O}(n + nnz(A))$. Wir brauchen Kriterien über die „Fülle“ eines Blocks.

5.1.2 Definition
Sei $G = (V, E)$ ein Graph, $V' \subseteq V$ und (V', E') der von V' induzierte Teilgraph von G. Dann ist die „Fülle“ $\varphi(V')$ von V' (bzgl. G) definiert als

$$\varphi(V') \;=\; \begin{cases} \frac{|E'|}{(|V'|^2 - |V'|)/2} & \text{falls } |V'| > 1 \\ 0 & \text{sonst .} \end{cases} \;.$$

Analog gilt für einen Digraphen $D = (V, E)$ und $V' \subseteq V$

$$\varphi(V') \;=\; \begin{cases} \frac{|E'|}{(|V'|^2 - |V'|)} & \text{falls } |V'| > 1 \\ 0 & \text{sonst .} \end{cases} \;.$$

$\varphi(V')$ gibt das Verhältnis der Kanten zur maximal möglichen Zahl (bei einer Clique mit $|V'|$ Knoten) an. Insbesondere gilt:

$$0 \leq \varphi(V') \leq 1.$$

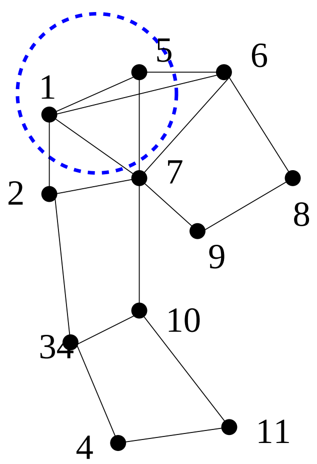
5.1.3 Definition
Ist $G = (V, E)$ ein Graph oder $D = (E, V)$ ein Digraph, $P \subset V$ und $v \in V \backslash P$. Dann erfüllt v das „Fülle-Kriterium“, wenn gilt

$$\varphi_{P \cup \{v\}} \geq \alpha \varphi_P \tag{5.1.1}$$

mit festem α. Die Zahl α heißt *Fülleparameter.*

Beachte: Je größer α, desto mehr Kanten hat der von P induzierte Teilgraph. Ein Wert $\alpha \geq 1$ ist denkbar und sinnvoll. Wenn v die Füllebedingung erfüllt, ist $\deg|_P(v)$ groß. Typisch: $\alpha = 1$.

5.1.4 Beispiel
Im Graphen



sei $P = \{1, 5\}$.
Knoten, die die Füllebedingung erfüllen mit

$\alpha = 1$: 6, 7

$\alpha = \frac{2}{3}$: 2, 6, 7

5.1.5 Definition
Sei $G = (V, E)$ ein Graph, $P \subset V$ und $v \in V$. Der *Grad (von v) in P*, notiert als $\deg|_P(v)$, ist der Grad von v im induzierten Teilgraphen $G|_{P\cup\{v\}}$. Für einen Digraphen D setzen wir

$$\begin{aligned} \deg_{\text{in}}|_P(v) &= \left|\left\{(w,v) \in E \;\middle|\; w \in P\right\}\right|, \\ \deg_{\text{out}}|_P(v) &= \left|\left\{(v,w) \in E \;\middle|\; w \in P\right\}\right| \\ \text{und} \qquad \deg|_P(v) &= \deg_{\text{in}}|_P(v) + \deg_{\text{out}}|_P(v). \end{aligned}$$

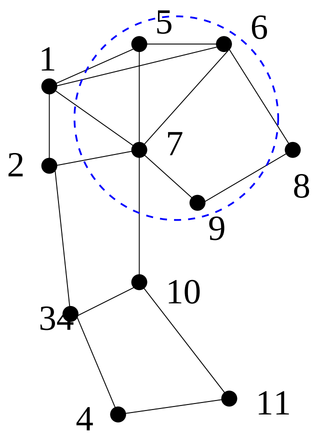
5.1.6 Definition
Sei $G = (V, E)$ ein Graph oder Digraph, $P \subset V$ und $v \in V \backslash P$ mit $\deg(v) > 0$. v erfüllt die *Verbundenheitsbedingung* falls

$$\frac{\deg|_P(v)}{\deg(v)} \geq \beta. \tag{5.1.2}$$

Die Zahl β heißt *Verbundenheitsparameter*.

Beachte: Je größer β, desto mehr Kanten von v gehen nach P, wenn v die Verbundenheitsbedingung erfüllt. Ein Wert $\beta > 1$ ist nicht sinnvoll. Die Verbundenheitsbedingung beruht auf relativen Werten, d.h. die absolute Größe $\deg(v)$ spielt keine Rolle. Typisch: $\beta = 0.5$.

5.1.7 Beispiel
Im Graphen



sei $P = \{5, 6, 7, 9\}$.
Knoten, die die Verbundenheitsbedingung erfüllen mit

$\beta = 1$: 8

$\beta = \frac{3}{4}$: 1

$\beta = \frac{1}{3}$: 2, 10

PABLO[1] (Parametrized Block Ordering) ist ein Algorithmus, der die Knoten eines Graphen in Gruppen (= Blöcke) zusammenfasst. Die Gruppen werden der Reihe nach aufgebaut. Knoten werden in eine Gruppe mit aufgenommen, wenn

- die Fülle-Bedingung erfüllt ist $\rightsquigarrow$ der Block bleibt dicht besetzt („innen viel“)
  oder

- die Verbundenheitsbedingung erfüllt ist $\rightsquigarrow$ es gibt nur wenige Eintäge in der Zeile/Spalte außerhalb des Blocks („außen wenig“)

Die Gruppenbildung entspricht einer Permutation π, PAP^T besitzt relativ volle Diagonalblöcke und relativ leere Außendiagonalblöcke.

[1] J. O'Neil and D. B. Szyld: A block ordering method for sparse matrices, SIAM J. Sci. Stat. Comput. **11**, 811-823 (1990)

5.1.8 Algorithmus
(PABLO für Graphen oder Digraphen (V, E))

```
Wähle α und β für (5.1.1) und (5.1.2)
C = V
setze Q := ∅ und P := ∅
while C ≠ ∅ do
   entnehme Knoten v aus C, füge ihn in P ein
   verschiebe alle zu v adjazenten Knoten aus C nach Q
   while Q ≠ ∅ do
      entferne einen Knoten w aus Q
      if w erfüllt (5.1.1) oder (5.1.2) then
         füge w in P ein
         verschiebe alle zu w adjazenten Knoten aus C nach Q
      else
         füge w in C ein
      end if
   end while
   entferne alle Knoten von P aus G, d.h. setze G := G|_C
   gebe P als nächsten Block aus
   set P := ∅
end while
```

Analyse von Algorithmus 5.1.8: Das Ergebnis des Algorithmus ist eine Permutation, gegeben durch die Reihenfolge, in welcher die Knoten in die Mengen P eingefügt werden. Jedes solche P ist ein Block.
Terminierung: $|V| := n$. Zu jedem Zeitpunkt ist $|Q| \leq n, |C| \leq n$ und $|C| + |Q| \leq n$. In jedem Schritt durch die äußere Schleife verringert sich $|C|$ um mindestens 1 $\rightsquigarrow$ terminiert. Jeder Durchlauf durch die innere Schleife verringert $|Q|$ oder $|C| + |Q|$. Dabei kann $|Q|$ wieder zunehmen, $|C| + |Q|$ jedoch nicht. Nach spätestens n^2 Schritten ist deshalb Q leer $\rightsquigarrow$ Schleife wird maximal n^2 Mal durchlaufen.
Aufwand: Wir gehen von folgenden Aufwänden $\mathcal{O}(1)$ aus (Realisierung mit geeigneten Datenstrukturen s.u.):

a) Entnehmen und Einfügen für C, Q, P: $\mathcal{O}(1)$

b) Aufdatierung von $\deg|_P(w)$ für alle $w \in adj(v)$: $\mathcal{O}(\deg(v))$

c) Aufdatierung von $\varphi(P)$ auf $\varphi(P \cup \{v\})$: $\mathcal{O}(\deg(v))$

d) Bestimmung aller zu v adjazenten Knoten: $\mathcal{O}(|adj(v)|)$

e) Bestimmung von $G := G|_{C'}$ aus G_C mit $C' = C \cup P$: $\mathcal{O}(|P| + |E'|)$, E' Kanten aus $G_{C'}$, die nicht in $G|_C$ sind.

5.1.9 Lemma
Unter diesen Voraussetzungen besitzt der PABLO-Algorithmus 5.1.8 den Aufwand $\mathcal{O}(|E| + |V|)$.

Beweis: Jeder Knoten w wird höchstens $\deg(w)$ mal in Q eingefügt, denn dazu muss er adjazent zu einem v aus P sein, und alle v kommen nur ein Mal nach P. Aufwand für Zeilen 6, 8, 10, 11 und 13 deshalb jeweils $\mathcal{O}(|E|)$. Aufwand für Zeile 5: $\mathcal{O}(|V|)$. Aufwand für Zeile 16: $\mathcal{O}(|V|+|E|)$, für Zeile 17: $\mathcal{O}(|V|)$. Fehlt noch der Aufwand für das Überprüfen von (5.1.1) und (5.1.2). Dieser ist jeweils $\mathcal{O}(1)$ plus Aufwand für die Aufdatierung von $\varphi(P)$ und $\deg|_P(v)$. Letzterer ist $\deg(v)$ und damit insgesamt (für alle v) $\mathcal{O}(|E|)$. $\square$

Wie erreicht man den Aufwand für die elementaren Operationen? Für Q, P genügt irgendeine lineare Datenstruktur wie Stack, Queue oder Liste, da die zu entnehmenden Knoten unspezifiziert sind. (Folge: PABLO verläuft unterschiedlich, je nach Wahl der Datenstruktur). Damit ist a) ok für Q, P. Aus C müssen wir vorgegebene Knoten mit Aufwand $\mathcal{O}(1)$ entfernen können (Zeile 6). Wir nehmen deshalb für C eine doppelt verkettete Liste mit „externem Index“, genau wie bei der Markowitz-Liste (Abschnitt 4.6): Bei Realisierung mit Feldern ist Knoten v an Stelle v gespeichert. $G = (V, E)$ nehmen wir in DCEL-Darstellung $\rightsquigarrow$ b), c), d), e) erfüllt.

Praktische Modifikationen:

a) Zu große Blöcke vermeiden: Setze Parameter *maxbs*. Wenn $|P| = maxbs$, innere Schleife abbrechen, Q nach C zurückschreiben, P ausgeben

b) Zu kleine Blöcke vermeiden: Setze Parameter *minbs*. Wenn $|P| < minbs$ nach Ende der inneren Schleife: Füge mehrere aufeinanderfolgende P in ursprünglich leere Menge P'; wenn $minbs \leq P'$ gib P' als nächsten Block aus. (Test auf zu große Böcke wird dann $|P'| + |P| = maxbs$). Sind P und P' Listen, ist das Vereinigen ein Aufwand $\mathcal{O}(1)$ (wenn man Listenanfang und -ende kennt).

5.1.10 Beispiel
Matrix `STEAM3` vom Matrix.Market[2]. Abbildungen 5.1 und 5.2 zeigen die Struktur der Matrix vor und nach der PABLO-Permutation ($\alpha = 1.0, \beta = 0.5$). Abbildung 5.3 zeigt den Digraphen mit PABLO-Nummerierung der Knoten.

[2]Ergebnisse der Diplomarbeit von David Fritzsche, Universität Wuppertal, 2004

Bemerkung:

- PABLO kümmert sich gar nicht um Diagonaleinträge $\rightsquigarrow$ bestimme zuerst eine Transversale, am Besten sogar eine Flaschenhals-Transversale.
- PABLO kümmert sich nicht darum, ob die Diagonalblöcke regulär sind, was für das Block-Jacobi-Verfahren wünschenswert wäre.

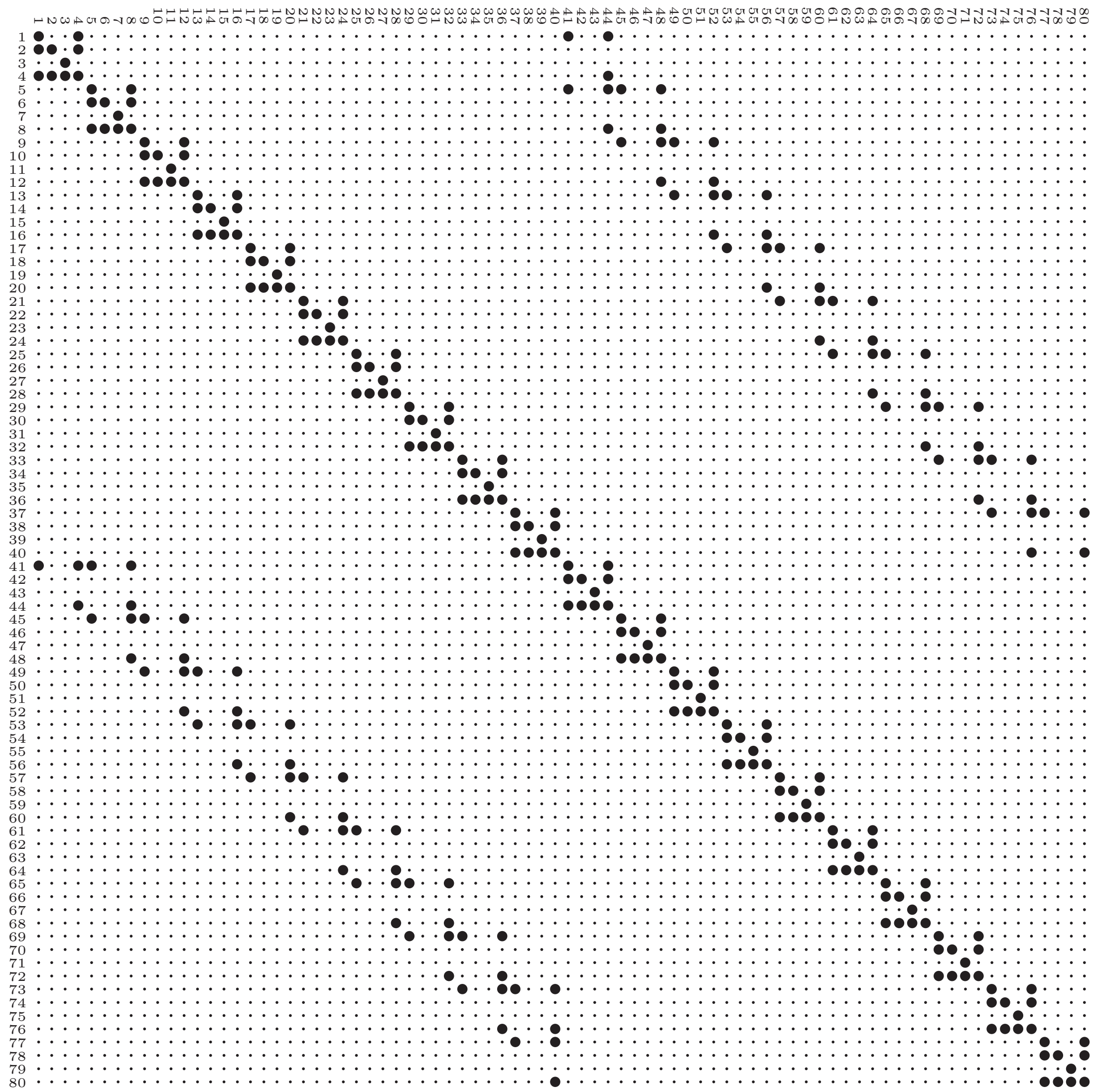


Abbildung 5.1: `STEAM3` Matrix, Original

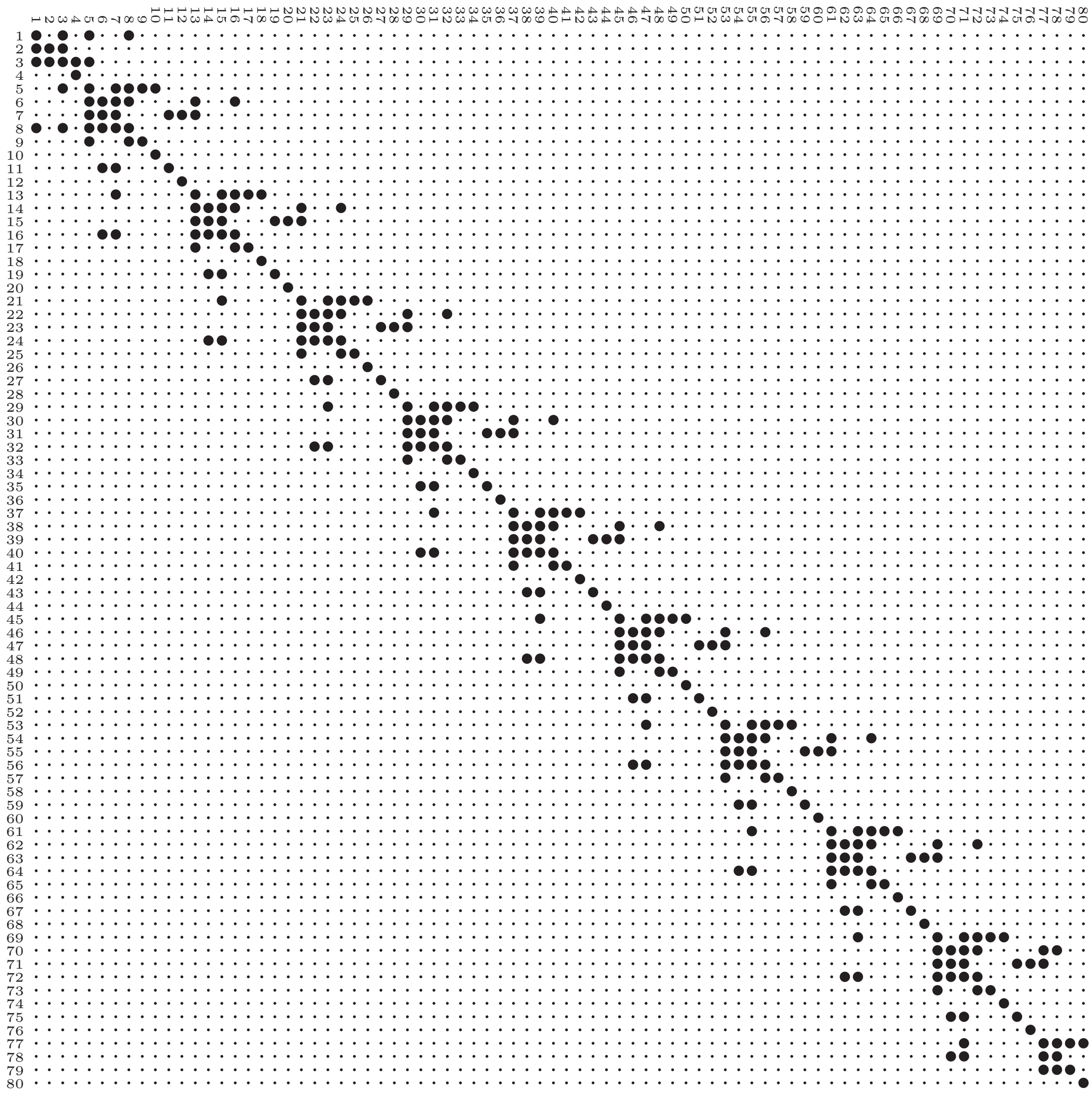


Abbildung 5.2: PABLO-permutierte `STEAM3` Matrix ($\alpha = 1.0$, $\beta = 0.5$)

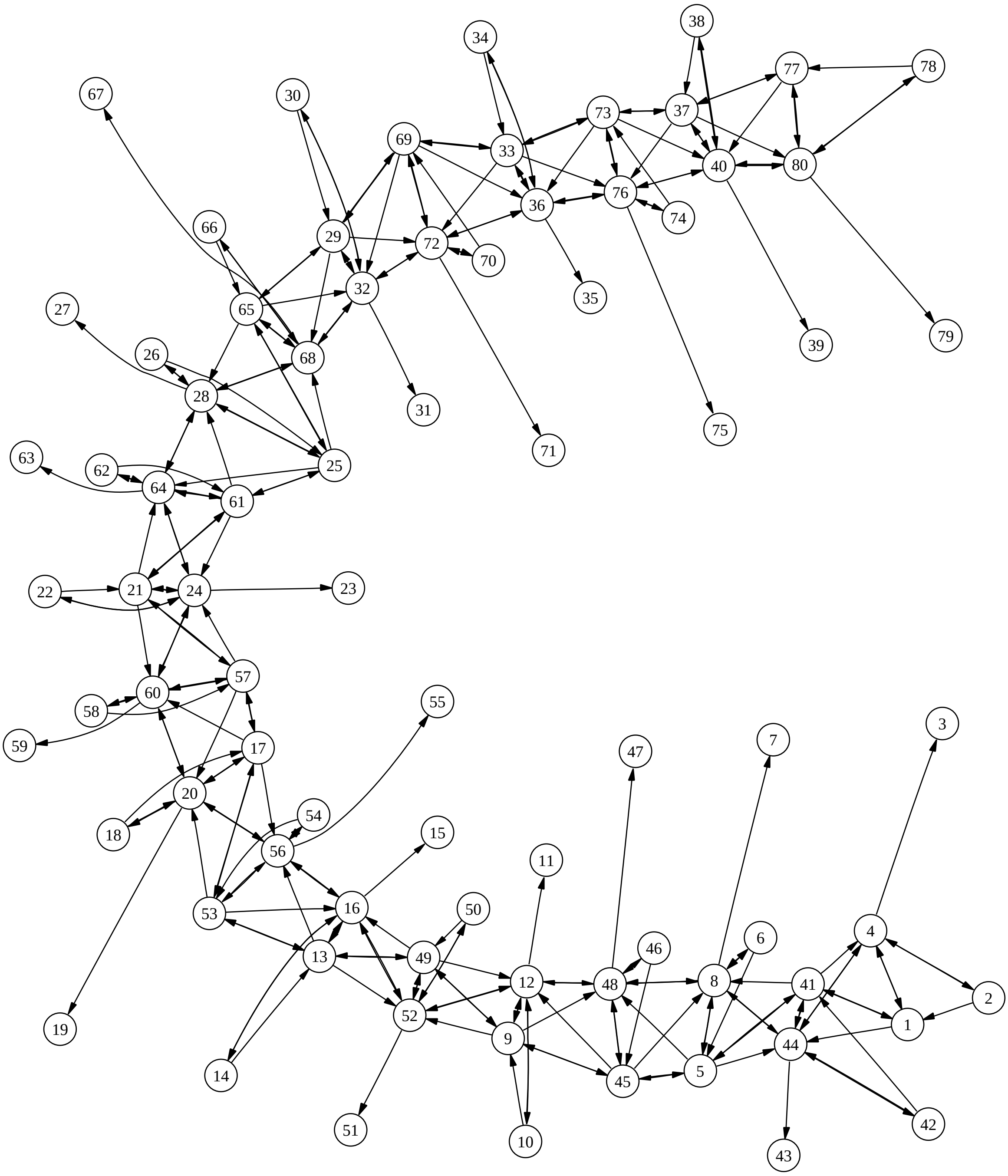


Abbildung 5.3: Digraph der `STEAM3` Matrix, PABLO-Nummerierung der Knoten

Abschnitt 5.2

Dichte Diagonalblöcke und große Einträge

Modifikation von PABLO: Berücksichtige nur solche Einträge (Kanten), die groß genug sind[3]. Neues Verfahren heißt TPABLO (T: threshold). Als Graphenproblem beschrieben hat man jetzt Kantengewichte $w(e)$.

5.2.1 Definition
Zu $A \in \mathbb{R}^{n\times n}$ gehört der kantengewichtete Digraph $G(A) = (V, E, w)$ mit V, E wi üblich und

$$w((i, j)) = |a_{ij}|.$$

Analog für den ungerichteten Graphen bei symmetrischen Matrizen.

5.2.2 Definition
Sei $G(A) = (V, E, w)$ der kantengewichtete Graph, $P \subseteq V$, $j \in V$. γ sei fest vorgegebender Parameter. Dann erfüllt j die TPABLO1-Bedingung, falls gilt

(5.2.1)
$$w((i, j)) = |a_{ij}| > \gamma \text{ oder } w((i, j)) = |a_{ji}| > \gamma \text{ für mindestens ein } i \in P.$$

Der Knoten j erfüllt die TPABLO2-Bedingung, falls gilt
(5.2.2)
$$w((i, j)) = |a_{ij}| > \gamma \text{ und } |w((i, j)) = a_{ji}| > \gamma \text{ für alle } i \in P,\ i \text{ adjazent zu } j.$$

Wir werden beide Bedingungen als Spezialfall einer allgemeineren Bedingung erleben.

5.2.3 Definition
Zu $A = (a_{ij})$ ist $A^{>\gamma}$ definiert als

$$A^{>\gamma} = (a_{ij}^{>\gamma}) \qquad \text{mit} \qquad a_{ij}^{>\gamma} := \begin{cases} a_{ij} & \text{falls } |a_{ij}| > \gamma, \\ 0 & \text{sonst.} \end{cases}$$

Entsprechend für gewichtete Graphen:

[3] H. Choi, D. B. Szyld: Application of Threshold Preconditioning of Sparse Matrices to Markov Chains, Proceedings of the IEEE International Computer Performance and Dependability Symposium IPDS'96, IEEE Computer Society Press, Los Alamitos, CA, 158-165 (1996)

5.2.4 Definition
$G = (V, E, w)$ sei ein gewichteter Graph und $\gamma \in \mathbb{R}$. Dann ist $G|^{>\gamma}$ der Teilgraph

$$G|^{>\gamma} := (V, E|^{>\gamma}, w) \quad \text{mit} \quad E|^{>\gamma} := \{(i,j) \in E \mid w((i,j)) > \gamma\}.$$

5.2.5 Lemma
Sei $A \in \mathbb{R}^{n\times n}$ und $G(A) = (V, E, w)$ mit $w((i,j)) = |a_{ij}|$. Dann ist

$$G(A^{>\gamma}) = G|^{>\gamma}.$$

Die folgende Idee kommt aus der Diplomarbeit David Fritzsche (2004).

5.2.6 Definition
Sei $A \in \mathbb{R}^{n\times n}$, $G(A)$ der Graph und $\gamma \in \mathbb{R}$. Dann ist

$$\deg|^{>\gamma}(p) \text{ im Graphen } G(A) := \deg(p) \text{ im Graphen } G(A^{>\gamma})$$

Entsprechend ist $\deg|_P^{>\gamma}$ erklärt.

5.2.7 Definition
$G(A) = (V, E)$ sei der Graph von $A \in \mathbb{R}^{n\times n}$, $P \subset V$, $p \in V \setminus P$ and $\gamma, \zeta \in \mathbb{R}$. Die *TPABLOx-Bedingung* lautet

$$\deg|_P^{>\gamma}(p) \geq \lceil \deg|_P(p) \cdot \zeta \rceil. \tag{5.2.3}$$

γ ist der *Schwellenparameter* und ζ der *Anteilsparameter*.

5.2.8 Lemma
Sei $G = (V, E)$ der Graph von A, $P \subset V$. Für alle $j \in V \setminus P$ und $\gamma \geq 0$ gilt

$$\deg_P^{>\gamma}(j) = \text{Anzahl der } i \in P \text{ mit } |a_{ij}| > \gamma \text{ oder } |a_{ji}| > \gamma$$

Mit der TPABLOx-Bedingung (5.2.3) bekommt man die TPABLO1-Bedingung (5.2.1) und die TPABLO2-Bedingung (5.2.2) unter einen Hut.

5.2.9 Lemma
(i) Für $0 < \zeta < 1/(2n-1)$ ist (5.2.3) äquivalent zu (5.2.1).

(ii) Für $\zeta = 1$ ist (5.2.3) äquivalent zu (5.2.2).

Zugehöriger Algorithmus: Modifikation von PABLO, wobei (5.2.3) zusätzlich gefordert wird, s. Zeile 9.

5.2.10 Algorithmus
(XPABLO für Graphen oder Digraphen (V, E))

Wähle α, β, γ und ζ für (5.1.1), (5.1.2) und (5.2.3)
$C = V$
setze $Q := \emptyset$ und $P := \emptyset$
while $C \neq \emptyset$ **do**
  entnehme Knoten v aus C, füge ihn in P ein
  verschiebe alle zu v adjazenten Knoten aus C nach Q
  while $Q \neq \emptyset$ **do**
    entferne einen Knoten w aus Q
    if w erfüllt ((5.1.1) oder (5.1.2)) und (5.2.3) **then**
      füge w in P ein
      verschiebe alle zu w adjazenten Knoten aus C nach Q
    else
      füge w in C ein
    end if
  end while
  entferne alle Knoten von P aus G, d.h. setze $G := G|_C$
  gebe P als nächsten Block aus
  set $P := \emptyset$
end while

Bemerkungen:

a) Wie bei PABLO sollte man zu kleine und zu große Blöcke verhindern.

b) Die TPABLO1-Bedingung versucht, außerhalb der Diagonalen alle Elemente $\leq \gamma$ zu halten.

c) Die TPABLO2-Bedingung versucht, in jedem Diagonbalblock alle (Nichtnull-) Einträge größer γ zu halten.

d) Es ist wichtig, A zunächst zu skalieren, denn γ ist eine absolute Größe. Mögliches Vorgehen: Bestimme Diagonalmatrix D_1 so, dass das betragsgrößte Element jeder Zeile den Betrag 1 hat: $A \to D_1 A$. Bestimme dann entsprechende Diagonalmatrix für die Spalten, $D_1 A \to D_1 A D_2$. Bestimme jetzt noch eine (Flaschenhals-) Transversale $A \to P D_2 A D_1$.

Ausblick: XPABLO bestimmt eine Blockdiagonalmatrix für ein Block-Jacobi-Verfahren. Verbesserung von Block-Jacobi, bei fast gleichem Aufwand: Nehme den Quotientengraphen $\mathcal{G}$ von $A + A^T$ bzgl. der Blockeinteilung. Bewerte alle Kanten in $\mathcal{G}$ (z.B. durch eine Norm des entsprechenden Matrix-Blocks). Finde maximal spannenden Baum in $\mathcal{G}$ (Algorithmus von Kruskal). Nehme Zerlegung $A = M - N$ mit M Matrix zum vom Baum induzierten Teilgraphen und Satz 3.5.2. (Diplomarbeit Fritzsche).

Abschnitt 5.3

Iterative Verbesserung

Wir schildern zum Abschluss eine neue Idee zur Permutation bei unsymmetrischen Systemen[4].
Ausgangspunkt: Trennung von symbolischer und numerischer Phase für unsymmetrische Systeme.
Ansatzpunkt: Treffe besondere Vorkehrungen, um Stabilität während der numerischen Phase zu verbessern.
Gerüst des Algorithmus

1: skaliere $A \to D_2 A D_1$
2: bestimme (Flaschenhals-) Transversale $A \to TA$
3: bestimme für den Graphen von $A + A^T$ oder $A^T A$ eine geeignete Permutation P, $A \to PAP^T$
4: faktorisere $A = LU$ mit Gauß-Elimination und Kontrolle über Pivot-Element:
5: **if** $|a_{ii}| < \sqrt{\epsilon}\|A\|$ **then**
6:     setze $a_{ii} = \sqrt{\epsilon}\|A\|$
7: **end if**
8: löse $Ax = b$ mittels iterativer Verbesserung:
9: setze $\eta' = 1,\ x = 0$
10: **repeat**
11:     $r = b - Ax$
12:     löse $A \cdot dx = r$ {approximativ über berechnete LU-Zerlegung}
13:     $\eta = \max_i \frac{|r_i|}{(|A| \cdot |x| + |b|)_i}$ {komponentenweiser Rückwärtsfehler}
14:     $x = x + dx,\ \eta' = \eta$
15: **until** $\eta < \epsilon$ oder $\eta \geq \eta'$

Hier ist ϵ die Maschinengenauigkeit. Es ist $A = LU - R$ mit (unbekanntem) R wegen Zeile 6. Deshalb wird dann mit dieser approximativen Zerlegung iteriert. Wichtig ist, dass man einen guten Test, nämlich bzgl. des komponentenweisen Rückwärtsfehlers verwendet. Dies diskutieren wir hier noch kurz.
Klassisches Resultat (normweiser Rückwärtsfehler, s. Numerik 1).

[4] X. Li, J. Demmel: Making Sparse Gaussian Elimination Scalable by Static Pivoting, Proceedings Supercomputing 98

5.3.1 Satz
Sei $\hat{x}$ eine Näherungslösung für $Ax = b$, $r = b - Ax$ und $\|\cdot\|$ eine Norm (und zugehörige Matrixnorm). Dann gilt: x ist exakte Lösung eines LGS $\hat{A}x = \hat{b}$ mit $\|A - \hat{A}\| < \rho\|A\|$, $\|b - \hat{b}\| \leq \rho\|b\|$ mit

$$\rho = \frac{\|r\|}{\|A\|\|\hat{x}\| + \|b\|}.$$

Normweise relative Rückwärtsfehler sind bei dünn besetzten Matrizen nicht so aussagekräftig: Man möchte den Zusammenhang mit der Lösung zu einem gestörten System mit derselben Struktur.

5.3.2 Definition
Sei $\hat{x}$ eine Näherungslösung für $Ax = b$. Der komponentenweise relative Rückwärtsfehler ω ist definiert als

$$\omega = \min\left\{\epsilon \mid (A + \Delta A)\hat{x} = b + \Delta b,\ |\Delta A| \leq \epsilon|A|\ |\Delta b| \leq \epsilon|b|\right\}.$$

5.3.3 Satz (Oettli-Prager)
Sei $\hat{x}$ eine Näherungslösung für $Ax = b$, $r = b - A\hat{x}$. Dann gilt:

$$\omega = \max_i \frac{|r_i|}{(|A|\hat{x} + |b|)_i}.$$