Speicher-Schemata für spärlich besetzte Matrizen

Peter Christen
Institut für Informatik
Universität Basel
Mittlere Strasse 142
CH-4056 Basel
email: christen@ifi.unibas.ch

16. September 1996

Zusammenfassung

Werden spärlich besetzte Matrizen (d.h. Matrizen mit kleinem Prozentsatz Elemente ungleich Null) mit speziellen Schemata gespeichert, kann einerseits der Speicheraufwand verringert werden, indem nur die Elemente ungleich Null gespeichert werden und andererseits die Rechenzeit für Matrix-Operationen verkürzt werden, indem nur mit diesen Elementen gerechnet wird.
In diesem Bericht werden verschiedene Speicher-Schemata für unstrukturierte spärlich besetzte Matrizen vorgestellt. Die Kriterien und Anforderungen an ein Speicher-Schema werden untersucht, ihr Speicheraufwand und ihre Zugriffszeiten werden analysiert und einige in Software-Paketen und in der Literatur aufgeführte Schemata kurz besprochen.

Inhaltsverzeichnis

1. Einleitung

Eine Matrix wird als spärlich besetzt (*sparse*) bezeichnet, wenn sie genügend viele Null-Elemente enthält, so dass es sich lohnt, spezielle Techniken anzuwenden, die das Speichern und Bearbeiten von Null-Elementen verhindern (Definition aus [19]). Alle Operationen auf einer Sparse-Matrix werden nur auf den Elementen ungleich Null (*Non-Zero-Elements, NZE*) durchgeführt.

Ein Speicher-Schema für Sparse-Matrizen besteht aus den gespeicherten Matrix-Elementen und einer administrativen Datenstruktur (Zeiger, Indizes, etc.). Der gesamthaft benötigte Speicher sollte möglichst klein sein. Die Speicherung einer vollbesetzten Matrix in einem zweidimensionalen Array (*Dense*-Speicherung) benötigt beispielsweise sehr wenig administrativen Aufwand, dafür werden auch Null-Elemente gespeichert.

Die Vorteile eines Speicher-Schemas für Sparse-Matrizen bestehen darin, dass weniger Speicher benötigt wird als bei der *Dense*-Speicherung und die Anzahl der arithmetischen Operationen von Matrix-Algorithmen verringert wird. Somit können grössere Matrizen bearbeitet werden. Nachteile hingegen sind: Es müssen komplexere Datenstrukturen für die Verwaltung der NZE aufgebaut werden und als Folge davon werden viele Matrix-Algorithmen komplizierter.

Die Wahl eines geeigneten Speicher-Schemas hängt – neben der Struktur der Matrix – im wesentlichen von den Operationen ab, die auf einer Matrix durchgeführt werden sollen. Oft sind verschiedene Algorithmen für eine bestimmte Operation vorhanden, so dass ein geeignetes Schema ausgewählt werden kann. Matrizen mit besonderer Struktur (z.B. symmetrische Matrizen, Band- oder Dreiecks-Matrizen) können jeweils mit speziellen Schemata gespeichert werden.

Für die Auswahl eines bestimmten Schemas sind folgende Kritierien zu berücksichtigen:

- Sind das Schema und die Algorithmen für alle Sparse Matrizen geeignet, oder ermöglichen sie nur die Bearbeitung von Matrizen mit bestimmten Strukturen?
- Sind effiziente Algorithmen für alle benötigten Operationen verfügbar? Falls nein, soll ein Algorithmus an ein Schema angepasst werden oder ein anderes Schema verwendet werden?
- Ist das Schema auf verschiedenen Architekturen (Parallel- und Vektorrechnern, RISC-Workstations) mit den jeweils verfügbaren Programmiersprachen effizient implementierbar?

Für vollbesetzte Matrizen sowie für Band-Matrizen wurden in den letzten Jahrzehnten Software-Pakete entwickelt, die in der Welt des wissenschaftlichen Rechnens weit verbreitet sind (z.B. LINPACK [4], LAPACK [2] und die BLAS-Routinen [6, 7]).
Es gibt zwar auch ähnliche Pakete für Sparse-Matrizen, diese sind jedoch nicht so erfolgreich wie die obengenannten. Ein Grund dafür mag sein, dass sich kein einheitliches Speicher-Schema durchgesetzt hat.

In diesem Bericht wird eine Übersicht von mehreren grundsätzlichen Speicher-Schemata für Sparse-Matrizen mit beliebiger Struktur gegeben. Beliebig bedeutet dabei, dass die NZE auf zufällige Weise über die ganze Matrix verteilt sein können, ohne Einschränkungen wie Symmetrie oder Bandcharakter.

In den folgenden Abschnitten werden die Anforderungen an ein Speicher-Schema, die Zugriffsstrukturen von grundlegenden Matrix-Operationen sowie Basisprozeduren, die ein Software-Paket für Sparse-Matrizen mit beliebiger Struktur enthalten sollte, beschrieben. In Kapitel 2 wird ein Vergleich von neun verschiedenen Schemata durchgeführt, und in Abschitt 2.4 werden diese mit den in Software-Paketen verwendeten bzw. in der Literatur beschriebenen Schemata verglichen. Anschliessend werden in Kapitel 3 der Speicheraufwand und die Zugriffszeiten der hier vorgestellten Schemata analysiert und Schlussfolgerungen gezogen.

1.1. Anforderungen an ein Speicher-Schema

Ein Speicher-Schema lohnt sich nur, falls es gegenüber der herkömmlichen *Dense*-Speicherung einer Matrix einen Vorteil bringt, wobei die Aspekte **Speicheraufwand** und **Zugriffszeiten** ausschlaggebend sind. Eine Matrix muss daher eine gewisse Grösse haben und ihre Besetztheit (Prozentsatz der NZE) darf nicht allzu gross sein.

Ein gutes Speicher-Schema für Sparse-Matrizen sollte folgende Anforderungen erfüllen:

- Kleiner Speicher-Overhead:
  Die Datenstruktur (Zeiger, Indizes, etc.) zur Verwaltung der NZE sollte möglichst wenig Speicher benötigen.

- Schneller Zugriff auf die gespeicherten NZE:
  Die Position eines NZE in der Datenstruktur sollte einfach berechnet bzw. ein NZE schnell gelesen und beschrieben werden können.

- Effiziente fundamentale Matrix-Algebra:
  Bei vielen Algorithmen erfolgen die Zugriffe zeilen- bzw. spaltenweise (Lesen des nächsten NZE in derselben Zeile bzw. Spalte). Solche Zugriffsstrukturen sollten effizient unterstützt werden. Bei gewissen Algorithmen ist zudem ein direkter Zugriff auf die Diagonalelemente von Vorteil.

- Dynamisches Einfügen von NZE in die Datenstruktur:
  Bei der Faktorisierung (z.B. LU-Zerlegung) einer Sparse-Matrix entstehen im allgemeinen NZE an Stellen, wo in der Original-Matrix *Null* stand. Diese neuen NZE können erst zur Laufzeit eingefügt werden.

- Das Schema kann effizient implementiert werden.

1.2. Zugriff auf Matrix-Elemente

Betrachtet man die grundlegenden algebraischen Operationen auf Matrizen bezüglich des Zugriffs auf die einzelnen Elemente, so ergeben sich unterschiedliche Anforderungen. Sehr oft wird zeilen- oder spaltenweise auf eine Matrix zugegriffen. Es lohnt sich deshalb, die Datenstruktur eines Schemas so aufzubauen, dass ein schneller zeilen- bzw. spaltenweiser Zugriff möglich ist.

- Matrix-Skalar-Multiplikation oder -Division
  Die gesamte Datenstruktur wird (sequentiell) durchlaufen und die Operation auf jedes gespeicherte NZE angewandt. Es werden keine neuen NZE erzeugt.

- Matrix-Skalar-Addition oder -Subtraktion
  Diese beiden Operationen müssen auf alle Elemente einer Matrix angewendet werden, also auch auf alle nicht gespeicherten Null-Elemente, die neu zu NZE werden. Von den bisherigen NZE können gewisse *Null* werden. Die Struktur der Sparse-Matrix geht verloren, die Datenstruktur muss daher neu aufgebaut werden bzw. die Matrix als *Dense*-Matrix (zweidimensionales Array) gespeichert werden.

- Matrix-Matrix-Addition oder -Subtraktion
  Elemente mit gleichen Indizes werden addiert bzw. subtrahiert. In der Ergebnismatrix wird ein Element nur dann gespeichert, falls das Resultat der Operation ungleich *Null* ist. NZE können in der Ergebnismatrix nur dort entstehen, wo in mindestens einer der beiden Operanden-Matrizen ein NZE stand.

- Matrix-Vektor-Multiplikation
  Jedes NZE einer Zeile wird mit dem entsprechenden Element des Vektors multipliziert und aufsummiert. Der Zugriff auf die Elemente der Matrix erfolgt zeilenweise ($x = A * b$) bzw. spaltenweise ($x = b^T * A$).

- Matrix-Matrix-Multiplikation
  Der Zugriff auf die eine Matrix erfolgt zeilenweise, auf die andere wird spaltenweise zugegriffen. Es existieren Algorithmen, welche auf beide Matrizen zeilenweise (siehe z.B. [14], Abschnitt 7.19) oder spaltenweise zugreifen.

- Transponierte einer Matrix
  Die Zeilen- und Spalten-Indizes der NZE werden vertauscht. Bei einigen Schemata kann dies recht einfach innerhalb der Datenstruktur erfolgen, bei anderen muss diese gesamthaft neu aufgebaut werden.

- Gauss-Elimination
  Bei diesem Algorithmus werden einerseits alle NZE einer Zeile bzw. Spalte durch einen Wert dividiert, andererseits wird ein Vielfaches einer Zeile bzw. Spalte zu einer anderen Zeile bzw. Spalte hinzuaddiert. Es entstehen in den meisten Fällen neue NZE.

1.3. Basisprozeduren für Sparse-Matrix

Ein Programm-Paket für Sparse-Matrizen sollte einerseits Prozeduren für algebraische Operationen und andererseits solche zur Verwaltung der Datenstruktur beinhalten. Minimal sollten folgende Prozeduren bereitgestellt werden:

- Initialisieren und Kopieren einer Sparse-Matrix:
  Eine neue, leere Datenstruktur sollte aufgebaut (Speicher-Allozierung und Variablen-Initialisierung) und eine schon bestehende Matrix kopiert werden können.

- Zugriff auf ein beliebiges Element:
  Beim Lesezugriff auf ein beliebiges Element sollte entweder der gespeicherte Wert zurückgegeben werden (falls das Element in der Datenstruktur gespeichert ist) oder $Null$, falls es nicht gespeichert ist (Null-Element).
  Beim Schreibzugriff kann entweder bereits ein NZE an der Stelle gespeichert sein oder ein neues NZE sollte eingefügt werden. Falls Einfügen nicht möglich ist (kein Einfügen zur Laufzeit erlaubt oder der allozierte Speicher ist voll), muss eine Fehlermeldung zurückgegeben werden. Sonst wird das neue NZE eingefügt und sein Wert gesetzt.
  Existiert schon ein NZE an der Stelle, muss nur der alte Wert überschrieben werden.

- Zugriff auf das nächste NZE in derselben Zeile bzw. Spalte:
  Bei vielen Algorithmen ist ein sequentieller Zugriff auf alle NZE einer Zeile bzw. Spalte nötig. Prozeduren, die solche Zugriffe schnell ermöglichen, können die Laufzeit eines Algorithmus beträchtlich verringern.

- Löschen einer Matrix:
  Die gesamte Datenstruktur einer Sparse-Matrix wird gelöscht und der Speicher wieder freigegeben.

- Lesen einer Matrix von Disk und Schreiben auf Disk:
  Es sollten verschiedene File-Formate gelesen und geschrieben werden können, z.B. das Harwell-Boeing- [10] und das MATLAB-Format [17].

- Ausgabe der Matrix-Struktur:
  Neben der Ausgabe aller gespeicherten NZE sind oft weitere Informationen über die Matrix erwünscht (Dimension, Anzahl NZE, maximale Anzahl NZE, etc.). Auch eine grafische Darstellung der Struktur einer Sparse-Matrix kann oft hilfreich sein.

- Sortieren und Überprüfen der Datenstruktur:
  Durch das Sortieren einer Datenstruktur kann je nach Speicher-Schema der zeilen- oder spaltenweise Zugriff beschleunigt werden. Später eingefügte Elemente werden beim Sortieren an ihre richtige Position in der Datenstruktur geschoben.
  Beim Überprüfen der Datenstruktur können einerseits eventuell gespeicherte Null-Elemente aus der Datenstruktur entfernt werden (z.B. alle NZE mit einem Wert kleiner als ein bestimmter Schwellenwert), andererseits können die Werte von Variablen geprüft und aktualisiert werden.

2. Vergleich

In diesem Kapitel werden mehrere Speicher-Schemata für unstrukturierte, nichtsymmetrische Sparse-Matrizen verglichen und deren Speicher- und Zugriffsaufwand angegeben.

Es wurden neun Schemata in *C++* unter *Unix* implementiert. Jede Implementation eines Schemas besteht aus einer Klasse *SpMatrix*, auf der (neben Erzeugen, Löschen, Sortieren, Ausdrucken aller gespeicherten NZE und Einlesen von einem File) die folgenden Prozeduren anwendbar sind[1]:

- `GetElem(↓i, ↓j, ↑val)`
  Lesen des Elementes `(i,j)` aus der Matrix. Der Wert wird in `val` zurückgegeben. Falls das Element nicht gespeichert ist, wird *Null* zurückgegeben.

- `SetElem(↓i, ↓j, ↑val, ↓newOK)`
  Schreiben bzw. Einfügen eines Elementes mit dem Wert `val` in die Matrix. Wenn `newOK` gesetzt ist, kann ein neues Element eingefügt werden (falls and der Stelle `(i,j)` noch kein Element vorhanden ist und genügend freier Speicherplatz zur Verfügung steht, sonst wird eine Fehlermeldung zurückgegeben). Ist an der Stelle `(i,j)` ein NZE gespeichert, wird es mit `val` überschrieben.

- `GetNextInRow(↓i, ↑j, ↑val, ↓Reset)`
  Mit dieser Prozedur können die NZE einer Matrix-Zeile sequentiell gelesen werden. Wenn `Reset` gesetzt ist, wird in `val` der Wert des ersten gespeicherten NZE (in der Zeile `i`) und sein Spaltenindex in `j` zurückgegeben; ansonsten der Wert des nächsten NZE. Ist kein weiteres NZE mehr in der Zeile `i` gespeichert, wird eine Fehlermeldung zurückgegeben.

- `GetNextInCol(↑i, ↓j, ↑val, ↓Reset)`
  Analog zur obigen Prozedur kann hier spaltenweise auf die Matrix zugegriffen werden.

Eine Sparse-Matrix wird im folgenden durch drei Parameter charakterisiert: m ist die Anzahl Zeilen, n die Anzahl Spalten und r gibt die Anzahl NZE an, welche in der Datenstruktur gespeichert sind.
Bei der Berechnung des benötigten Speicherplatzes wird davon ausgegangen, dass die gespeicherten NZE Gleitkommazahlen und die Indizes Integerzahlen sind. In den Formeln für den Speicheraufwand sind mit *Daten* die Werte der NZE, mit *Index* ihre Indizes und mit *Pointer* Zeiger benannt. Arrays und die Nummerierung der Zeilen und Spalten einer Matrix beginnen mit Index *Null*.

Da die Zugriffszeiten von der Implementation abhängen, wird nur die durchschnittliche Anzahl Vergleiche (bis ein Element in der Datenstruktur gefunden ist) und Zuweisungen (bis es eingefügt bzw. der Wert geschrieben ist) für einen Aufruf der entsprechenden Prozedur angegeben

[1] ↓ bezeichnet Eingabe-, ↑ Ausgabeparameter

Zur Veranschaulichung wird jeweils die Datenstruktur der folgenden Sparse-Matrix dargestellt ($m = 5, n = 6, r = 9$):

$$\begin{pmatrix} 0.0 & -5.3 & 0.0 & 0.0 & 13.1 & 0.0 \\ 0.0 & 0.0 & 0.0 & -2.4 & 0.0 & -1.1 \\ 0.0 & 29.1 & 0.0 & 0.0 & -7.6 & 0.0 \\ 0.0 & 0.0 & 31.9 & 0.0 & 0.0 & 0.0 \\ 42.0 & 0.0 & 0.0 & 0.0 & 91.5 & 0.0 \end{pmatrix}$$

2.1. Statische Arrays

Speicher-Schemata bestehend aus Arrays werden vor allem in Programmen verwendet, die in *FORTRAN* geschrieben sind (z.B. SPARSKIT [16]).

Der Vorteil der Schemata mit Arrays ist, dass ihr Speicher-Overhead kleiner ist als bei den Schemata mit verketteten Listen.
Ihr Nachteil ist jedoch, dass bei den meisten Programmiersprachen die Grösse des benötigten Speichers schon beim Kompilieren bekannt sein muss bzw. die Array-Grenzen festgelegt werden müssen. Ein Array muss an einem Stück alloziert werden. Es kann später nicht erweitert, sondern muss in einen neu allozierten, grösseren Speicherbereich umkopiert werden. Zur Laufzeit können somit nicht beliebig viele Elemente eingefügt werden.

2.1.1. Koordinaten-Schema mit drei Arrays (KS-3A)

Dies ist das einfachste aller Schemata. Es werden drei Arrays der Länge r benötigt. Im ersten (`value`) werden die NZE gespeichert, im zweiten die Zeilen-Indizes (`rowIndex`) und im dritten die Spalten-Indizes (`colIndex`).
Das Array `value` enthält Gleitkomma-, die beiden anderen Integerzahlen. Die Datenstruktur muss nicht sortiert sein, die Zugriffszeit kann jedoch durch Sortieren verringert werden.
Dieses Schema wird von vielen Software-Paketen verwendet, vor allem zur externen Speicherung von Matrizen auf Disk.

Speicheraufwand	$r * (Daten + 2 * Index)$			
Zugriffsaufwand	Sortiert		Unsortiert	
	Vergleiche	Zuweisungen	Vergleiche	Zuweisungen
GetElem	$log_2(r)$	1	$r/2$	1
SetElem	$log_2(r)$	1	$r/2$	1
SetElem - Insert	$log_2(r) + 1$	3	$r/2 + 1$	3
GetNextInRow - Reset	$log_2(r) + 1$	2	r	$2 * r/(2 * m)$
GetNextInRow - No Reset	2	2	r	$2 * r/(2 * m)$
GetNextInCol - Reset	$log_2(r) + 1$	2	r	$2 * r/(2 * n)$
GetNextInCol - No Reset	2	2	r	$2 * r/(2 * n)$

value	42.0	31.9	-5.3	29.1	-2.4	13.1	91.5	-7.6	-1.1
rowIndex	4	3	0	2	1	0	4	2	1
colIndex	0	2	1	1	3	4	4	4	5

2.1.2. Zwei Arrays mit berechneten Positionen (BP-2A)

Dies ist eine Abwandlung des **KS-3A** Schemas, bei dem die beiden Index-Arrays zum Array `posIndex` vereinigt werden. Die Werte in `posIndex` berechnen sich folgendermassen: $posIndex[p] = (n * i + j), 0 \leq p < r$, wobei i und j die Indizes eines NZE sind. Wird die Datenstruktur nach aufsteigendem p sortiert, so ist die Matrix zeilenweise gespeichert. Analog kann die Matrix auch spaltenweise gespeichert werden, wenn $posIndex[p] = (i + m * j)$ ist und wiederum nach p sortiert wird. Die Datenstruktur muss nicht sortiert sein, die Zugriffszeit kann jedoch durch Sortieren verringert werden.
Die beiden Arrays des Schemas haben die Länge r, wobei `value` Gleitkomma- und `posIndex` Integerzahlen enthält.

Dieses Schema wird von SPARSKIT [16] und IMSL [20] unterstützt, von Chenyi Hu et al. werden in [11] Zugriffs-Messungen besprochen.

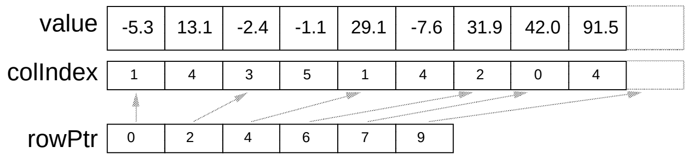
Speicheraufwand	$r * (Daten + Index)$			
Zugriffsaufwand	Sortiert		Unsortiert	
	Vergleiche	Zuweisungen	Vergleiche	Zuweisungen
GetElem	$log_2(r)$	1	$r/2$	1
SetElem	$log_2(r)$	1	$r/2$	1
SetElem - Insert	$log_2(r) + 1$	3	$r/2 + 1$	3
GetNextInRow - Reset	$log_2(r) + 1$	2	r	$2 * r/(2 * m)$
GetNextInRow - No Reset	2	2	r	$2 * r/(2 * m)$
GetNextInCol - Reset	$log_2(r) + 1$	2	r	$2 * r/(2 * n)$
GetNextInCol - No Reset	2	2	r	$2 * r/(2 * n)$

value	-5.3	13.1	-2.4	-1.1	29.1	-7.6	31.9	42.0	91.5
posIndex	1	4	9	11	13	16	20	24	28

2.1.3. Gepacktes Schema mit drei Arrays (GS-3A)

Dieses Schema ist auch unter dem Namen *Yale Sparse Matrix Format* oder *Compressed Sparse Row (CSR)* bekannt und wird in vielen Software-Paketen verwendet [9, 15, 16].
Es besteht aus drei Arrays. `value` enthält die NZE und `colIndex` ihre entsprechenden Spalten-Indizes, die Einträge von `rowPtr` zeigen jeweils auf den Beginn einer Zeile in `colIndex`.
Die Werte in `rowPtr` berechnen sich nach folgender Formel: $rowPtr[0] = 0, rowPtr[i] = rowPtr[i-1] + y(i-1), 0 \leq i < m$, wobei $y(i)$ die Anzahl der gespeicherten NZE in der Zeile i ist. Das letzte Element in `rowPtr` zeigt eine Position hinter das letzte belegte Element des Arrays `colIndex`. Die Matrix wird zeilenweise gespeichert, innerhalb einer Zeile müssen die NZE aber nicht sortiert sein.
Das Array `value` enthält r Gleitkommazahlen, `colIndex` und `rowPtr` enthalten r bzw. $m + 1$ Integerzahlen.

Speicheraufwand	$(m+1)*Index + r*(Daten + Index)$			
Zugriffsaufwand	Sortiert		Unsortiert	
	Vergleiche	Zuweisungen	Vergleiche	Zuweisungen
GetElem	$log_2(r/m)$	1	$r/(2*m)$	1
SetElem	$log_2(r/m)$	1	$r/(2*m)$	1
SetElem - Insert	$log_2(r/m)+1$	$(2*r+m)/2+3$	$r/(2*m)+1$	$(2*r+m)/2+3$
GetNextInRow - Reset	1	2	r/m	$2*r/(2*m)$
GetNextInRow - No Reset	1	2	r/m	$2*r/(2*m)$
GetNextInCol - Reset	$(m*n)/r*log_2(r/m)$	2	$(m*n)/r*r/(2*m)$	2
GetNextInCol - No Reset	$(m*n)/r*log_2(r/m)$	2	$(m*n)/r*r/(2*m)$	2

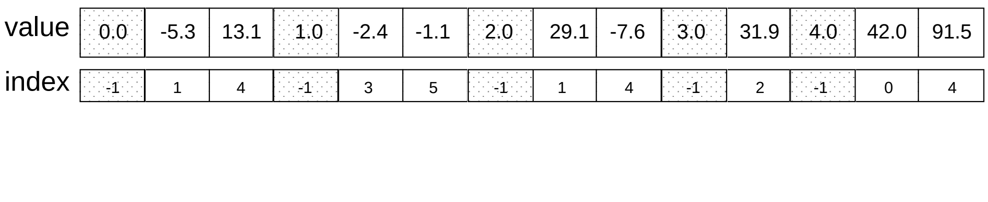


Analog kann die Matrix auch spaltenweise gespeichert werden (*Compressed Sparse Column (CSC)* genannt), wobei dann die Arrays `rowIndex` und `colPtr` benötigt werden.

2.1.4. Zwei Arrays mit Dummy-Elementen (DE-2A)

Beschrieben wird dieses Schema in [12]. Es besteht aus zwei Arrays: In `index` werden die Spalten-Indizes der NZE gespeichert, in `value` ihre Werte. Zudem geben *Dummy-Elemente* mit dem Spalten-Index -1 jeweils den Beginn einer neuen Zeile an. Im zugehörigen Eintrag in `value` steht der Zeilen-Index der folgenden Elemente (eine Umwandlung Gleitkomma- in Integerzahl ist notwendig!). Innerhalb einer Zeile müssen die Elemente nicht sortiert sein.
Die Speicherung erfolgt zeilenweise. Analog könnte die Matrix auch spaltenweise gespeichert werden, die Dummy-Elemente würden in diesem Fall den Beginn einer neuen Spalte markieren. Durch Sortieren der Zeilen bzw. Spalten kann der zeilen- bzw. spaltenweise Zugriff beschleunigt werden.

Speicheraufwand	$(r+m)*(Daten + Index)$			
Zugriffsaufwand	Sortiert		Unsortiert	
	Vergleiche	Zuweisungen	Vergleiche	Zuweisungen
GetElem	$(r+m+r/m)/2$	1	$(r+m+r/m)/2$	1
SetElem	$(r+m+r/m)/2$	1	$(r+m+r/m)/2$	1
SetElem - Insert	$(r+m+r/m)/2+1$	$2*(r+m)/2+3$	$(r+m+r/m)/2+1$	$2*(r+m)/2+3$
GetNextInRow - Reset	$(r+m)/2+1$	2	$(r+m+r/m)/2+1$	2
GetNextInRow - No Reset	2	2	$r/(2*m)$	2
GetNextInCol - Reset	$(m*n)/r*(r+m)/(2*m)$	2	$(m*n)/r*(r+m)/(2*m)$	2
GetNextInCol - No Reset	$(m*n)/r*(r+m)/(2*m)$	2	$(m*n)/r*(r+m)/(2*m)$	2



2.2. Verkettete Listen

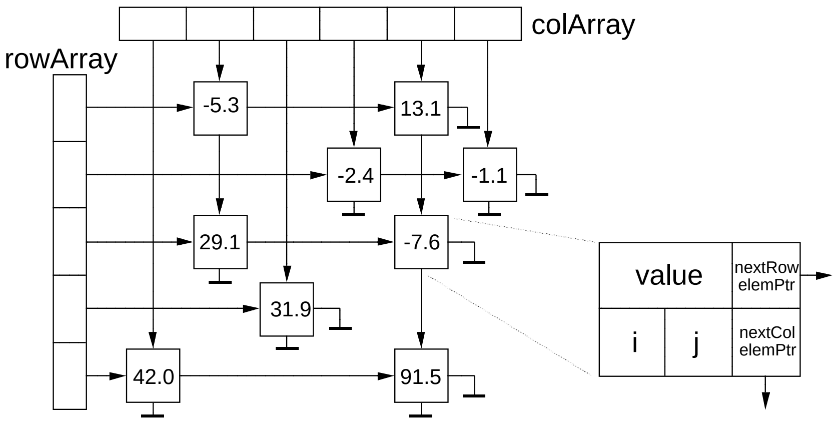
Verkettete Listen haben den grossen Vorteil, dass zur Laufzeit neue Elemente eingefügt und alte gelöscht werden können. Der benötigte Speicher wird erst zur Laufzeit alloziert.
Ihr Nachteil ist, dass der Speicher-Overhead gross ist. Pro Element wird mindestens ein Zeiger auf das nächste Element in einer Liste benötigt, bei mehrfach verketteten Listen sind es mehrere Zeiger. Auch besteht keine Möglichkeit, direkt auf ein Element zuzugreifen. Eine Liste muss sequentiell durchsucht werden, bis das gewünschte Element gefunden wird. Abhilfe schaffen Hilfs-Zeiger, die auf das zuletzt zugegriffene Element zeigen, so dass direkt das nächste Element erreicht werden kann.

2.2.1. Zwei Arrays mit zweidimensional verketteten Listen (2A-2DL)

Zwei Arrays (`rowArray[0..m-1]` und `colArray[0..n-1]`) enthalten Zeiger auf die Listen mit den Daten-Elementen. Eine 'horizontale' Liste verbindet alle NZE einer Zeile, eine 'vertikale' alle einer Spalte. Jedes Daten-Element besteht aus dem Wert (`value`), seinen beiden Indizes (`i` und `j`) sowie zwei Pointern zu den nächsten Elementen in derselben Zeile bzw. Spalte (`nextRowElemPtr` und `nextColElemPtr`).
Pro Element werden eine Gleitkommazahl, zwei Integerzahlen und zwei Zeiger gespeichert.

Das Softwarepaket *Sparse* [13] verwendet diese Datenstruktur. Zusätzlich wird darin ein Array von Zeigern auf alle Diagonal-Elemente verwaltet.

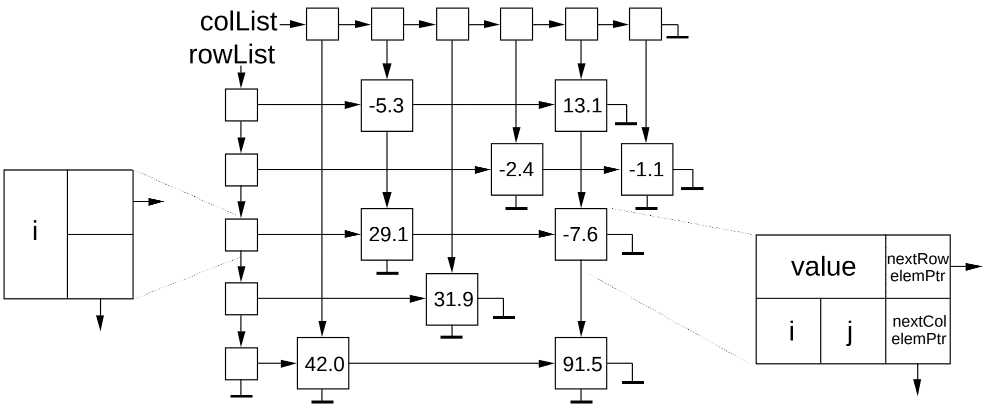
Speicheraufwand	$r * (Daten + 2 * Index + 2 * Pointer) + (n + m) * Pointer$	
Zugriffsaufwand	Sortiert	
	Vergleiche	Zuweisungen
GetElem	$min(r/m, r/n)/2$	1
SetElem	$min(r/m, r/n)/2$	1
SetElem - Insert	$min(r/m, r/n)/2 + max(r/m, r/n)/2$	5
GetNextInRow - Reset	1	2
GetNextInRow - No Reset	1	2
GetNextInCol - Reset	1	2
GetNextInCol - No Reset	1	2



2.2.2. Zwei Listen mit zweidimensional verketteten Listen (2L-2DL)

Die beiden Arrays des **2A-2DL** Schemas werden durch zwei verkettete Listen ersetzt (`rowList` und `colList`). Jedes Element dieser Listen enthält den Zeilen- bzw. Spalten-Index und zwei Zeiger. Der eine zeigt auf das nächste Element der Liste und der andere auf das erste Daten-Element in der entsprechenden Zeile bzw. Spalte. Die Daten-Elemente haben den gleichen Aufbau wie beim **2A-2DL** Schema.
Diese Datenstruktur ermöglicht dynamisches Einfügen und Löschen von Zeilen und Spalten.

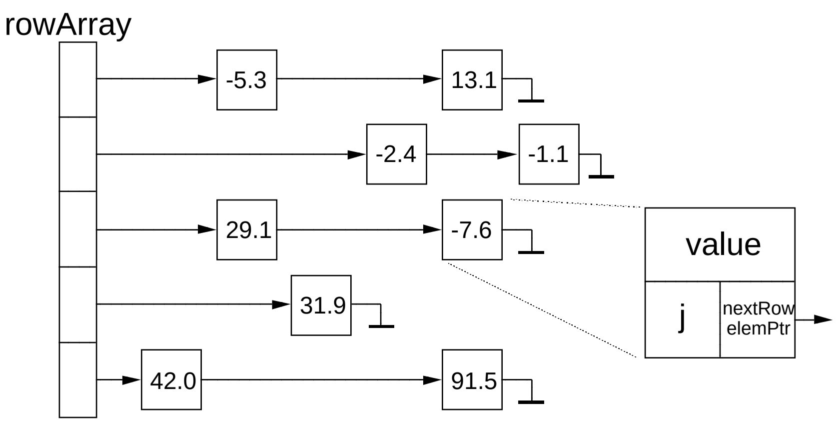
Speicheraufwand	$r * (Daten + 2 * Index + 2 * Pointer) + (n + m) * (2 * Pointer + Index)$	
Zugriffsaufwand	Sortiert	
	Vergleiche	Zuweisungen
GetElem	$min(m + r/m, n + r/n)/2$	1
SetElem	$min(m + r/m, n + r/n)/2$	1
SetElem - Insert	$min(m + r/m, n + r/n)/2 + max(m + r/m, n + r/n)/2$	5
GetNextInRow - Reset	$m/2 + 1$	2
GetNextInRow - No Reset	1	2
GetNextInCol - Reset	$n/2 + 1$	2
GetNextInCol - No Reset	1	2



2.2.3. Ein Array mit einfach verketteten Zeilen-Listen (1A-1DL)

Dies ist eine vereinfachte Version von **2A-2DL** mit nur zeilenweise verketteten Listen. Das Array `rowArray[0..m-1]` enthält pro Eintrag einen Zeiger auf eine Liste mit allen Daten-Elementen einer Zeile. Pro Daten-Element wird der Wert (`value`), der Spalten-Index (`j`) und ein Zeiger (`nextRowElemPtr`) auf das nächste Daten-Element gespeichert (pro Element ein Zeiger, eine Gleitkomma- und eine Integerzahl). Der Speicher-Overhead ist, verglichen mit dem **2A-2DL** Schema, einiges geringer.
Analog könnte man die Listen auch spaltenweise verketten, aus Symmetriegründen betrachten wir hier nur die zeilenweise Version.

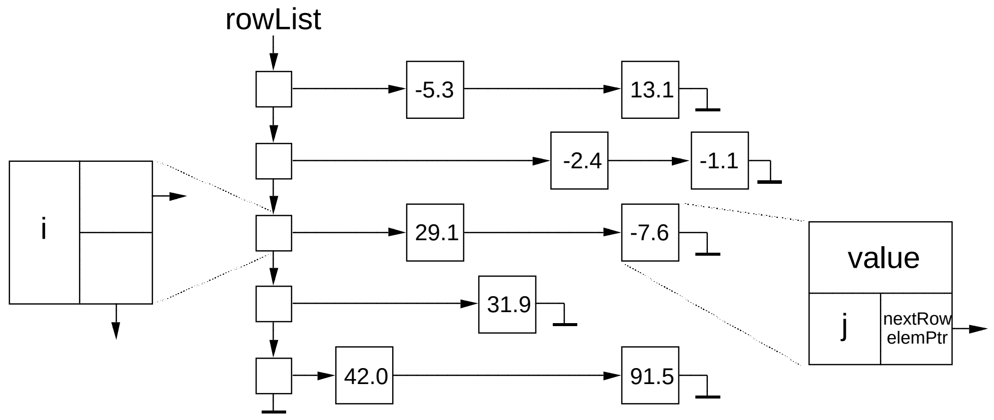
Speicheraufwand	$r * (Daten + Index + Pointer) + m * Pointer$	
Zugriffsaufwand	Sortiert	
	Vergleiche	Zuweisungen
GetElem	$r/(2*m)$	1
SetElem	$r/(2*m)$	1
SetElem - Insert	$r/(2*m)$	3
GetNextInRow - Reset	1	2
GetNextInRow - No Reset	1	2
GetNextInCol - Reset	$(m*n)/r*r/(2*m)$	2
GetNextInCol - No Reset	$(m*n)/r*r/(2*m)$	2



2.2.4. Eine Liste mit einfach verketteten Zeilen-Listen (1L-1DL)

Anstelle des Arrays des vorangehenden Abschnitts wird hier – analog zu **2L-2DL** – eine Liste (`rowList`) mit Zeigern auf die Zeilen-Listen verwendet. Ein Element dieser Liste besteht aus dem Zeilen-Index und zwei Zeigern. Der eine zeigt zum nächsten Element in der Liste, der andere zur Zeilen-Liste mit den Daten-Elementen. Die Daten-Elemente sind gleich aufgebaut wie beim **1A-1DL** Schema.
Auch diese Datenstruktur kann spaltenweise aufgebaut werden.

Speicheraufwand	$r * (Daten + Index + Pointer) + m * (2 * Pointer + Index)$	
Zugriffsaufwand	Sortiert	
	Vergleiche	Zuweisungen
GetElem	$m/2 + r/(2*m)$	1
SetElem	$m/2 + r/(2*m)$	1
SetElem - Insert	$m/2 + r/(2*m)$	3
GetNextInRow - Reset	$m/2 + 1$	2
GetNextInRow - No Reset	1	2
GetNextInCol - Reset	$(m*n)/r*r/(2*m)$	2
GetNextInCol - No Reset	$(m*n)/r*r/(2*m)$	2

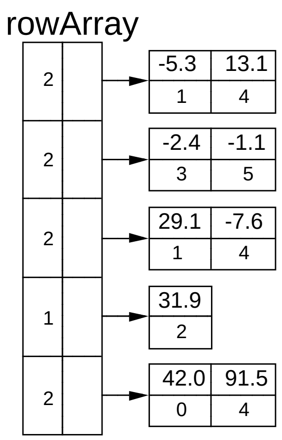


2.2.5. Ein Array mit Zeigern auf Zeilen-Arrays (1A-ZA)

Ein Array (`rowArray`) der Länge m enthält Zeiger auf Daten-Arrays, in denen jeweils alle NZE einer Zeile mit ihren Spalten-Indizes gespeichert sind. Im `rowArray` steht zudem jeweils die Anzahl der gespeicherten NZE in der entsprechenden Zeile.
Jeder Eintrag in den Daten-Arrays besteht aus dem Wert des NZE, als Gleitkommazahl gespeichert, und dem Spalten-Index (Integer). Das `rowArray` enthält pro Eintrag einen Zeiger und eine Integerzahl.

Eine Abwandlung dieses Schemas wird von der *Meschach Library* [18], einem australischen Software-Paket, verwendet. Zusätzlich sind dort die Spalten verkettet.

Speicheraufwand	$m * (Index + Pointer) + r * (Daten + Index)$			
Zugriffsaufwand	Sortiert		Unsortiert	
	Vergleiche	Zuweisungen	Vergleiche	Zuweisungen
GetElem	$log_2(r/m)$	1	$r/(2*m)$	1
SetElem	$log_2(r/m)$	1	$r/(2*m)$	1
SetElem - Insert	$log_2(r/m)+1$	3	$r/(2*m)+1$	3
GetNextInRow - Reset	1	2	r/m	$2*r/(2*m)$
GetNextInRow - No Reset	1	2	r/m	$2*r/(2*m)$
GetNextInCol - Reset	$(m*n)/r*log_2(r)$	2	$(m*n)/r*r/(2*m)$	2
GetNextInCol - No Reset	$(m*n)/r*log_2(r)$	2	$(m*n)/r*r/(2*m)$	2



2.3. Ergänzungen

Die Besprechung der neun Schemata erhebt keinen Anspruch auf Vollständigkeit. Es sind einige in der Literatur beschriebene und in der Praxis verwendete Schemata. Oft werden in einem Software-Paket modifizierte Formen dieser Schemata verwendet.

Bei vielen Sparse-Matrizen sind alle Diagonalelemente ungleich *Null*. Falls viele Zugriffe auf diese Elemente vorgenommen werden müssen, können sie in einem separaten Array gespeichert werden. Jedes hier vorgestellte Schema kann so modifiziert werden, teilweise werden solch modifizierte Schemata auch von Software-Paketen unterstützt [16].

Bei der praktischen Implementierung wird meist eine Anzahl zusätzlicher Variablen mit in die Datenstruktur aufgenommen:

- Die Grösse des allozierten und des momentan belegten Speichers.
- Die aktuelle sowie die maximal zulässige Zahl der NZE (falls kein beliebiges dynamisches Einfügen möglich ist).
- Die Zahl der NZE, die während eines Algorithmus (z.B. LU-Zerlegung) erzeugt und gespeichert werden (sog. *Fill-In*).
- Eine Array mit Zeigern auf die Diagonal-Elemente.
- Eine boolsche Variable, die angibt ob die Datenstruktur sortiert ist.
- Der Name der Matrix (wenn sie z.B. aus der Harwell-Boeing-Kollektion [10] stammt).
- Eine Hilfs-Datenstruktur zum Speichern von Zwischenresultaten.
- Die Konditionszahl der Matrix.

2.4. Speicher-Schemata in Software-Paketen und in der Literatur

Es gibt heute eine grosse Anzahl von kommerziellen oder Public-Domain Software-Paketen für Sparse-Matrizen. Da es kein standardisiertes Speicher-Schema gibt, verwendet jedes Paket sein eigenes Format; abhängig von der Programmiersprache, in der das Paket geschrieben wurde, Verwendungszweck oder Zielarchitektur (Workstations, Vektor- oder Parallelrechner).

Es folgt eine kurze Vorstellung von Software-Paketen und Büchern:

- Die Autoren von *SPARSKIT* [16] versuchen mit ihrem Paket eine Art Standard – ähnlich wie LAPACK für vollbesetzte Matrizen – für Anwendungen mit Sparse-Matrizen aufzubauen.
  Es bietet unter anderem Routinen zur Konvertierung und Bearbeitung von 16 verschiedenen Speicher-Schemata an. Von den hier vorgestellten Schemata werden folgende von *SPARSKIT* unterstützt: **KS-3A**, **BP-2A** und **GS-3A** (in einer zeilen - und spaltenorientierten Form). Weiterhin werden mehrere Schemata für Band- und Block-Matrizen angeboten. Das Paket ist in FORTRAN geschrieben und kostenlos erhältlich.

- Auch das *Sparse BLAS Toolkit* [3] soll als grundlegender Standard dienen, ähnlich wie die BLAS-Routinen für Dense-Matrizen. Ein Standard muss nach Meinung der Autoren viele verschiedene Speicher-Schemata für alle gängigen Algorithmen unterstützen.
  In der aktuellen Version werden vom *Sparse BLAS Toolkit* folgende Schemata unterstützt: **KS-3A**, **GS-3A** (zeilen- und spaltenweise) sowie mehrere Schemata für Diagonal- und Block-Matrizen. Falls von den Anwendern gewünscht, sollen in Zukunft noch weitere Schemata hinzugefügt werden. Auch die Zahl der angebotenen Operationen (bisher nur Matrix-Matrix-Multiplikation, Lösen eines Dreiecks-Gleichungssystems, Permutation einer Sparse-Matrix und Überprüfen der Datenstruktur) soll erweitert werden.

- *Sparse 1.3* [13] ist eine Bibliothek von C-Routinen zum Bearbeiten von grossen Sparse-Matrizen, wobei reelle und komplexe Matrizen verarbeitet werden können. Die verwendete Datenstruktur entspricht dem hier vorgestellten **2A-2DL** Schema, jedoch erweitert mit einem Array von Zeigern, die auf die Diagonal-Elemente einer Matrix zeigen. Zusätzlich wird eine grosse Anzahl von Informationen über eine Matrix mitgespeichert (z.B. Anzahl der Fill-In Elemente, Grösse der Matrix im Speicher, Pivotisierungsart, etc.), insgesamt etwa 50 Variablen. Das Paket ist Public Domain über `netlib` [5] erhältlich.

- Die *Meschach Library* [18] ist ein anderes in C geschriebenes Paket, das eine Erweiterung für Sparse-Matrizen enthält. Das verwendete Schema entspricht einem erweiterten **1A-ZA** Schema, wobei die Diagonal-Elemente direkt – über den separat gespeicherten Array-Index des Diagonal-Elementes – zugreifbar sind. Eine spaltenweise Verkettung der NZE kann ebenfalls erreicht werden, denn jeder Eintrag in den Daten-Arrays hat einen Verweis auf das NZE in derselben Spalte in der nächsten Zeile. Auch dieses Paket ist frei über `netlib` erhältlich.

- Die objektorientierte Klassen-Bibliothek *SparseLib++* wird in [8] und [15] vorgestellt. Mit dem objektorientierten Ansatz sollen dem Benutzer Implementierungsdetails wie das Speicher-Schema verborgen bleiben. Das *Sparse BLAS Toolkit* [3] wird als Kern verwendet, damit das Paket auf viele Architekturen portiert werden kann.
  Drei Schemata werden unterstützt (**KS-3A**, **GS-3A** zeilen- und spaltenweise). Für jedes wird eine Klasse aufgebaut, mit der Matrizen erzeugt, gelöscht, geladen, gespeichert und auf die Elemente zugegriffen werden kann. Einfügen von neuen Elementen ist bei keinem Schema möglich.
  Weiterhin werden bisher drei Möglichkeiten zur Präkonditionierung unterstützt (Diagonal, Incomplete Cholesky und Incomplete LU), jede als eigene Klasse implementiert. Eine Erweiterung des Paketes ist geplant.

- Das *Sparse Matrix Manipulation System (SMMS)* [1] bietet einzelne Algorithmen bzw. Teilschritte von Algorithmen als Unix- und DOS-Kommandos (also ausführbare Programme) an. Matrizen werden als ASCII Textdateien im Koordinaten-Schema (**KS-3A**) gespeichert. Die einzelnen Kommandos benutzen diese als Eingabe und speichern ihre Resultate wiederum in eine ASCII Textdatei. Somit kann eine Matrix nach jedem ausgeführten Kommando betrachtet werden. Dieses Paket ist vor allem für die Lehre gedacht.

- IMSL [20] ist eine kommerziell vertriebene Software-Bibliothek für viele Anwendungen der Mathematik, Statistik und den Ingenieurswissenschaften. Sie ist in FORTRAN- und C-Versionen erhältlich. Für Sparse-Matrizen wird das **KS-3A** Schema verwendet.

- Auch *MATLAB* [17] kann Sparse-Matrizen bearbeiten, es stehen mehrere Befehle für ihre Bearbeitung zur Verfügung. Extern werden Sparse-Matrizen im **KS-3A** Schema gespeichert, wobei in einem Eintrag in den Indizes die Dimension der Matrix steht, im entsprechenden Eintrag in `value` eine *Null*.

- In *An Analysis of Sparse Matrix Storage Schemes* [19] bespricht M. Veldhost das Software-Paket *TORRIX-SPARSE*, das zwei neue, vom Autor entwickelte, Schemata benutzt.
  Beim ersten Schema wird die Matrix zeilenweise gespeichert, wobei jeweils zwischen dem ersten und dem letzten NZE einer Zeile alle Elemente (auch Null-Elemente) gespeichert werden. Durch Permutieren der Matrix kann die Anzahl der gespeicherten Null-Elemente verringert werden. Dieses Schema eignet sich für Dreiecks- und Bandmatrizen.
  Das zweite Schema beruht auf einer baumförmigen Datenstruktur. Eine Matrix wird rekursiv in Blöcke und Unterblöcke aufgeteilt. Ein Unterblock wird als vollbesetzte oder als Band-Matrix gespeichert. Auf die optimale Aufteilung einer Matrix und die Anwendbarkeit des Schemas für die LU-Zerlegung wird ausführlich eingegangen. Das in diesem Buch vorgestellte Software-Paket basiert auf diesem baumartigen Schema.
  Es werden auch die generellen Aspekte von Speicher-Schemata besprochen, wie Zugriffsstrukturen von verschiedenen numerischen Algorithmen (direkte und iterative Solver für lineare Gleichungssysteme, Eigenwert- und Least-Square-Probleme) und die unterschiedlichen Strukturen von Sparse-Matrizen, wie sie bei praktischen Anwendungen (z.B. Differential-Gleichungen oder Optimierungs-Problemen) entstehen.
  Alle in diesem Buch entwickelten und abgedruckten Programme (sowie das gesamte *TORRIX-SPARSE* - Paket) sind in Algol 68 geschrieben.

- S. Pissanetzky erläutert in *Sparse Matrix Technology* [14] ebenfalls mehrere Schemata. Da alle Algorithmen in diesem Buch in FORTRAN geschrieben sind, basieren die Schemata auf Arrays. Neben Band- und Block-Schemata werden das **GS-3A**-Schema und eine Version von **2A-2DL** vorgestellt, bei der die zweidimensional verketteten Listen durch Arrays ersetzt wurden.

- In *Direct Methods for Sparse Matrices* [9] von I.S. Duff et al. werden mehrere, in FORTRAN-Notation beschriebene, Schemata vorgestellt. Begonnen wird mit dem Koordinaten Schema (**KS-3A**). Das **GS-3A** Schema wird als 'Kollektion von Sparse-Vektoren' bezeichnet und Array-Versionen des **1A-1DL** und **2A-2DL** werden ebenfalls besprochen. Auf die Anwendbarkeit von grundlegenden Operationen (Vektor-Vektor-Addition, Matrix-Vektor-Multiplikation) mit den verschiedenen Schemata wird kurz eingegangen.

3. Analyse

3.1. Speicheraufwand

In der folgenden Tabelle wird der Speicheraufwand der neun präsentierten Schemata verglichen. Zusätzlich wird der Speicheraufwand bei *Dense*-Speicherung aufgeführt (d.h. falls alle Elemente einer Matrix in einem zweidimensionalen Array gespeichert werden).

Die Dimension der Matrix ist durch m Zeilen und n Spalten gegeben. Die Anzahl der NZE in der Datenstruktur sei r. Die Besetztheit (d.h. der Prozentsatz der NZE im Verhältnis zu allen Elementen) wird mit s *(Sparsity)* angegeben.

Es wird angenommen, dass Pointer und Indizes eine Länge von 4 Byte haben, die NZE werden im *Double*-Format (8 Bytes) gespeichert. Diese Werte gelten für die meisten Unix-Workstations.

m (Zeilen)	100	1'000	100	1'000	100	1'000	1'000	10'000
n (Spalten)	100	100	1'000	100	1'000	1'000	1'000	10'000
r (NZE)	1'000	1'000	1'000	10'000	10'000	10'000	100'000	100'000
s (% NZE)	10%	1%	1%	10%	10%	1%	10%	0.1%
Dense	80'000	800'000	800'000	800'000	800'000	8'000'000	8'000'000	800'000'000
KS-3A	16'000	16'000	16'000	160'000	160'000	160'000	1'600'000	1'600'000
GS-3A	12'404	16'004	12'404	124'004	120'404	124'004	1'204'004	1'240'004
BP-2A	12'000	12'000	12'000	120'000	120'000	120'000	1'200'000	1'200'000
DE-2A	13'200	24'000	13'200	132'000	121'200	132'000	1'212'000	1'320'000
2A-2DL	24'800	28'400	28'400	244'400	244'400	248'000	2'408'000	2'480'000
2L-2DL	26'400	37'200	37'200	253'200	253'200	264'000	2'424'000	2'640'000
1A-1DL	16'400	20'000	16'400	164'000	160'400	164'000	1'604'000	1'640'000
1L-1DL	17'200	28'000	17'200	172'000	161'200	172'000	1'612'000	1'720'000
1A-ZA	12'800	20'000	12'800	128'000	120'800	128'000	1'208'000	1'280'000

Tabelle 1. Vergleich des Speicheraufwands (in Byte)

Je nach Dimension und Besetztheit der Matrix wird der Speicher-Overhead der Datenstruktur unterschiedlich gross. Die untenstehende Tabelle zeigt auf, wie gross der minimale und maximale Overhead für die obenstehenden Beispiel-Matrizen ist[1].

Schema	KS-3A	GS-3A	BP-2A	DE-2A	2A-2DL	2L-2DL	1A-1DL	1L-1DL	1A-ZA
Overhead	100	51 - 100	50	52 - 200	201 - 255	203 - 365	101 - 150	102 - 250	51 - 150

Tabelle 2. Mimimaler und maximaler Speicher-Overhead (in Prozent)

[1] 100% Overhead besagt, dass die gesamte Datenstruktur doppelt so gross ist, als wenn nur die Matrix-Elemente gespeichert würden.

Der Overhead bei den Array-Schemata ist kleiner als bei dynamischen Datenstrukturen. Je nach Verhältnis von m zu n ändert sich der Overhead bei einigen Schemata, weil die Grösse der Datenstruktur von der Anzahl der Zeilen (bzw. Spalten) abhängt (z.B. **GS-3A** und **DE-2A**). Der Overhead bei dynamischen Datenstrukturen variiert sehr stark.

3.2. Zugriffszeiten

In diesem Abschnitt werden die Laufzeiten für die am Anfang von Kapitel 2 vorgestellten Prozeduren besprochen und verglichen.
Es wurden zwei Testserien mit zufällig besetzten, quadratischen ($m = n$) Matrizen durchgeführt. Bei der ersten Serie wurde die Dimension von 50 bis 2000 variiert, wobei jeweils 1% der Elemente ungleich $Null$ waren. Für die zweite Serie wurden Matrizen der Dimension 500 verwendet, ihre Besetztheit wurde von 0.2% auf 10% gesteigert.

Testserie 1: Variation der Dimension			**Testserie 2:** Variation der Besetztheit		
Dimension (m, n)	Anzahl NZE (r)	Besetztheit	Dimension (m, n)	Anzahl NZE (r)	Besetztheit
50	25	1 %	500	500	0.2%
100	100	1 %	500	1'000	0.4%
250	625	1 %	500	2'500	1%
500	2'500	1 %	500	5'000	2%
1'000	10'000	1 %	500	10'000	4%
2'000	40'000	1 %	500	25'000	10%

Tabelle 3. Verwendete Testmatrizen

Folgende Prozeduren wurden getestet:

- `GetElem`
  Jedes Element einer Matrix wurde einzeln gelesen, unabhängig davon, ob es gespeichert war oder nicht (insgesamt $m * n$ Lesezugriffe).

- `SetElem - Insert`
  r NZE wurden in eine anfangs leere Datenstruktur eingefügt. Bei den **KS-3A** und **BP-2A** Schemata wurden die neuen NZE am Schluss der (unsortierten) Datenstruktur eingefügt, bei allen anderen Schemata an die entsprechende Stelle in der sortierten Datenstruktur.

- `SetElem`
  r bereits gespeicherte NZE wurden mit neuen Werten beschrieben, aber keine neuen NZE in die Matrix gespeichert (kein Einfügen).

- `GetNextInRow - Reset=1`
  Es wurde das erste gespeicherte NZE in jeder Zeile gelesen (insgesamt m Zugriffe).

- `GetNextInRow - Reset=0`
  Die restlichen $r - m$ NZE in der Datenstruktur wurden zeilenweise gelesen. Der Zugriff auf das erste NZE in jeder Zeile wurde bei der Zeitmessung nicht mitgerechnet.

- `GetNextInCol - Reset=1`
  Jeweils das erste gespeicherte NZE einer Spalte wurde gelesen (n Zugriffe).

- `GetNextInCol - Reset=0`
  Die restlichen $r - n$ gespeicherten NZE wurden spaltenweise gelesen.

Pro Prozedur wurde jeweils die Laufzeit von 1000 Tests gemessen und die durchschnittliche Zeit für einen Aufruf (z.B. ein `GetElem`) berechnet. Die Zugriffszeit in den folgenden Grafiken ist jeweils in μ-Sekunden angegeben. Alle Messungen wurden auf einer HP Workstation (Modell 712/80) mit 32 MByte Hauptspeicher durchgeführt.

3.2.1. Testserie 1: Variation der Dimension

Lesezugriff auf einzelne Elemente: `GetElem`

Die Zugriffszeit beim **DE-2A** Schema nimmt sehr stark zu, da die zu durchsuchende Datenstruktur mit der Anzahl der NZE wächst und nur lineares Durchsuchen möglich ist. Die Zeit für **KS-3A** und **BP-2A** erhöht sich weniger schnell, da bei diesen Schemata die Datenstruktur binär durchsucht werden kann. **GS-3A** und **1A-ZA** sind noch schneller, weil nur die NZE einer Zeile binär durchsucht werden.
2L-2DL und **1L-1DL** werden mit zunehmender Dimension sehr langsam, da die `rowList` und die Liste mit den NZE einer Zeile nur linear durchsucht werden können. **1A-1DL** und **2A-2DL** sind viel schneller, da direkter Zugriff auf eine Zeile (bzw. Spalte) möglich ist (unabhängig von der Dimension) und nur die NZE einer Zeile linear durchsucht werden müssen.
Die Zugriffszeiten des **Dense**-Schemas bleibt praktisch konstant. Ab der Dimension 2000 beanspruchte die Datenstruktur der Testmatrizen jedoch mehr Speicher als von der Workstation zur Verfügung gestellt wurde; Teile der Matrizen mussten ausgelagert werden (*Swapping*), was die Ausführungszeit der Tests drastisch erhöhte.

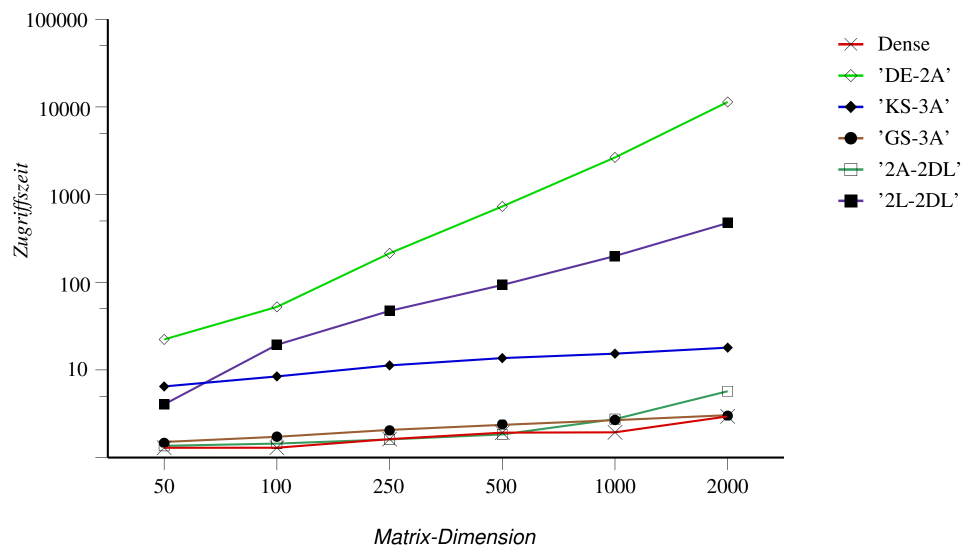


Abbildung 1. Zugriffszeiten `GetElem`

Schreibzugriff auf einzelne Elemente: `SetElem`

Da bei diesem Test die Datenstruktur des **KS-3A** Schemas nicht sortiert war, konnte nur lineares Suchen (wie bei **DE-2A**) angewendet werden, daher ist die Zunahme der Zugriffszeiten bei diesen beiden Schemata fast gleich. Die Zeiten von **GS-3A** nehmen nur ganz leicht zu, da sich die Zahl der NZE pro Zeile (die durchsucht werden muss) nur wenig erhöht und zudem binär gesucht wird.
Auch beim **2A-2DL** Schema muss nur eine Zeile durchsucht werden, die Zeiten erhöhen sich jedoch etwas stärker, da nur linear gesucht werden kann. Bei **2L-2DL** ist die Zunahme noch grösser, da hier zusätzlich die `rowList` linear durchsucht wird.
Beim **Dense**-Schema machen sich ab der Dimension 2000 wiederum die Auswirkungen des *Swappens* bemerkbar.

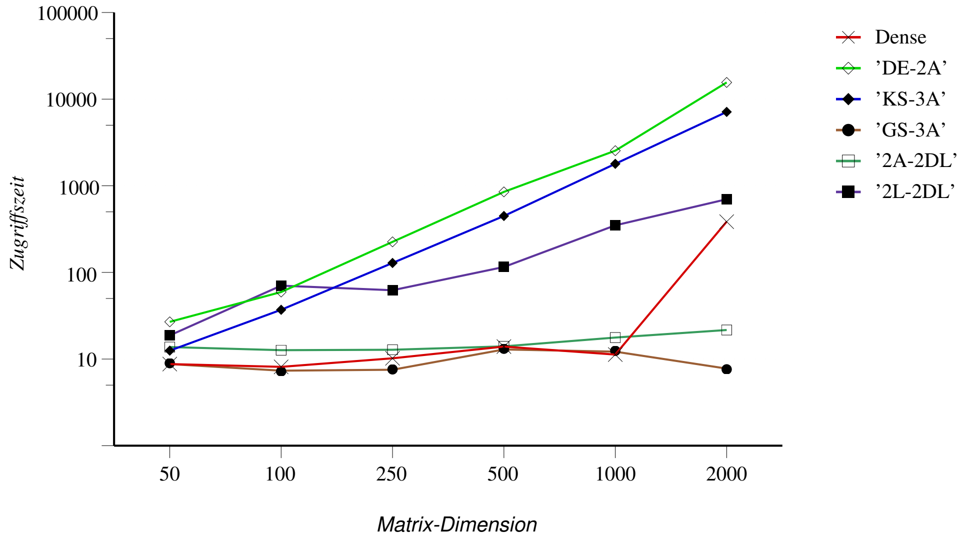


Abbildung 2. Zugriffszeiten `SetElem`

Einfügen von NZE: `SetElem - Insert`

Die Zeiten von **KS-3A** und **BP-2A** sind gleich und steigen stark an, da die Datenstruktur linear durchsucht und am Schluss eingefügt wird. Auch **DE-2A** wird linear durchsucht, jedoch an der richtigen Stelle eingefügt und alle folgenden NZE um eine Position nach hinten geschoben.
Beim **GS-3A** Schema erhöht sich die Zugriffszeit ebenfalls relativ stark, da auch hier an der richtigen Stelle eingefügt wird (die Datenstruktur ist immer sortiert) und alle NZE hinter der Einfügestelle um eins nach hinten geschoben werden. Das Schema **1A-ZA** ist viel schneller, da nur innerhalb einer Zeile nach hinten geschoben werden muss.
Die Zeiten von **2L-2DL** erhöhen sich stark, da hier beim Einfügen einerseits die `rowList` (zum Finden der gewünschten Zeile) und die `colList` (zum Finden der Spalte) und dann die Zeilen- und Spaltenlisten linear duchsucht werden müssen (das neue NZE muss in beide Listen eingefügt werden). **2A-2DL** ist schneller, da direkt auf die Zeile und Spalte zugegriffen werden kann.
Die Einfüge-Zeit beim **Dense**-Schema bleibt fast konstant, bis die Matrix grösser als der Speicher ist und sich das *Swappen* bemerkbar macht.

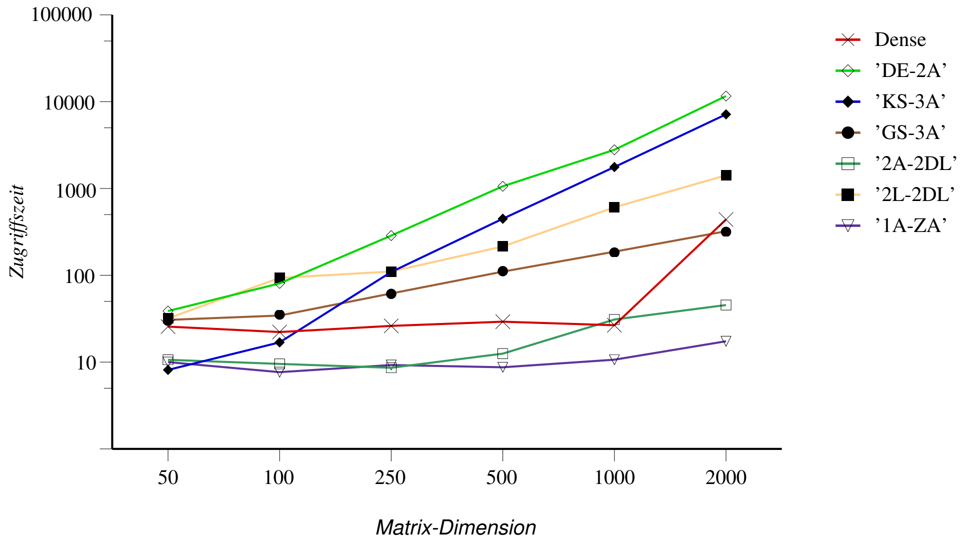


Abbildung 3. Zugriffszeiten `SetElem - Insert`

Zugriff auf die ersten NZE einer Zeile: `GetNextInRow - Reset=1`

Die Zugriffszeiten für die Schemata **1A-1DL**, **2A-2DL**, **1A-ZA** und **GS-3A** bleiben konstant schnell, unabhängig von der Dimension der Matrix, da direkt auf die ersten NZE einer Zeile zugegriffen werden kann. Bei **KS-3A** und **BP-2A** nimmt die Zeit langsam zu, da die Datenstrukturen binär durchsucht werden. **DE-2A** kann nur linear durchsucht werden, die Zugriffszeiten nehmen dementsprechend stark zu.
Bei **2L-2DL** und **1L-1DL** nehmen die Zeiten mit steigender Dimension ebenfalls sehr stark zu, da hier jeweils die `rowList` linear durchsucht wird, bis die entsprechende Zeile gefunden ist.
Die Zunahme beim **Dense**-Schema lässt sich dadurch erklären, dass bei grösseren Matrizen immer mehr Null-Elemente durchsucht werden müssen, bis das erste NZE in einer Zeile gefunden wird.

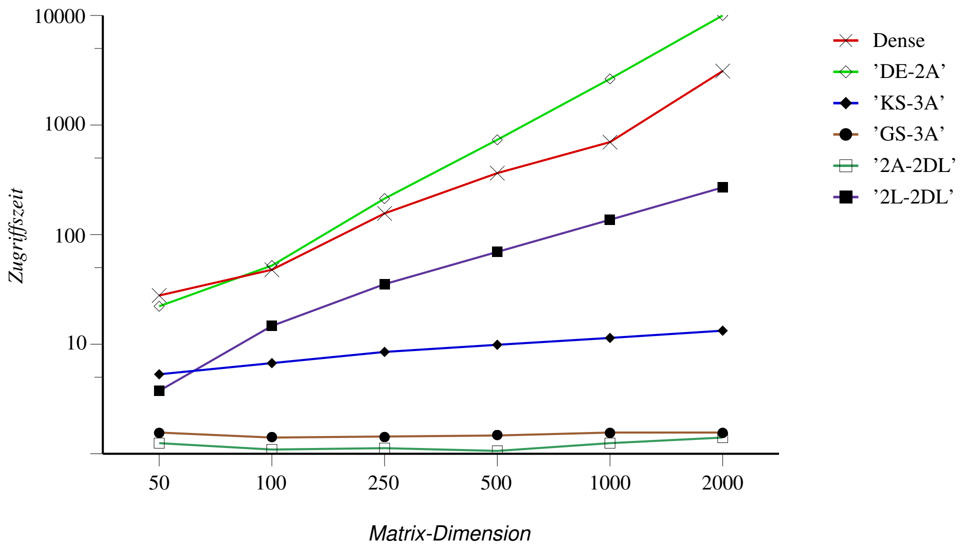


Abbildung 4. Zugriffszeiten `GetNextInRow - Reset=1`

Zugriff auf weitere NZE einer Zeile: `GetNextInRow - Reset=0`

Da alle Array-Schemata (**BP-2A**, **DE-2A**, **KS-3A**, **GS-3A** und auch **1A-ZA**) zeilenweise sortiert sind, kann direkt auf das nächste NZE in einer Zeile zugegriffen werden, unabhängig von der Dimension der Matrix.
Bei den Schemata mit verketteten Listen (**1A-1DL**, **2A-2DL**, **1L-1DL** und **2L-2DL**) ist ebenfalls direkter Zugriff auf das nächste NZE in derselben Zeile möglich, die Zeiten erhöhen sich jedoch für grosse Matrizen und mehr NZE ein wenig, da öfters ein *Cache-Miss* vorkommt, d.h. öfters ein neuer Bereich des Hauptspeichers in den Cache geladen werden muss.
Die leicht grösseren Zugriffszeiten für ganz kleine Matrizen lassen sich durch das Testprogramm erklären: Die Initialisierung einer While-Schleife (die alle NZE einer Zeile ausliest) braucht einige Zeit, was sich bei wenigen NZE pro Zeile bemerkbar macht, bei vielen NZE jedoch verschwindet.
Die Zugriffszeit beim **Dense**-Schema ist ungefähr konstant, da bei 1% Besetztheit durchschnittlich jedes hundertste Element der Matrix ungleich *Null* ist, und somit – unabhängig von der Dimension – durchschnittlich 99 Null-Elemente durchsucht werden müssen.

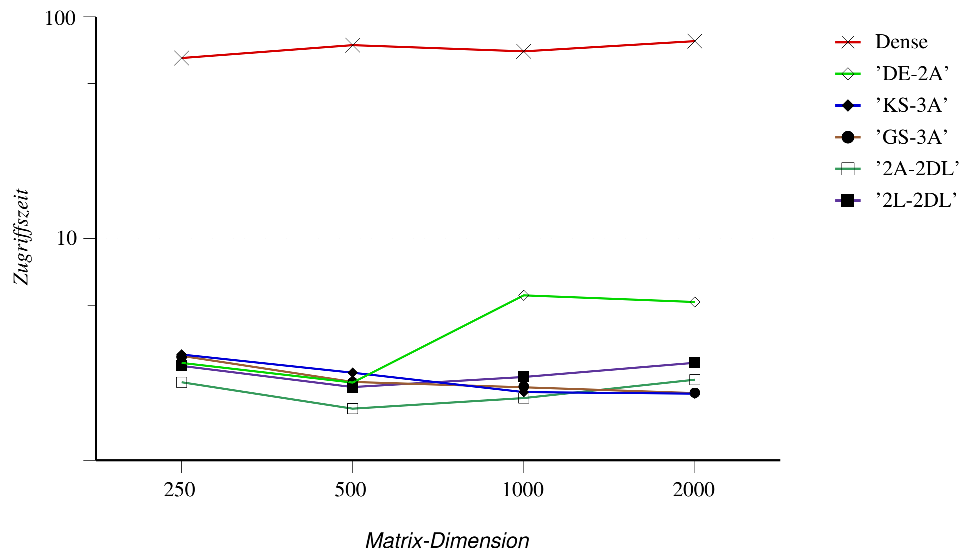


Abbildung 5. Zugriffszeiten `GetNextInRow - Reset=0`

Zugriff auf die ersten NZE einer Spalte: `GetNextInCol - Reset=1`

Mit Ausnahme von **2A-2DL** steigen die Zugriffszeiten bei allen Schemata sehr stark an, da kein direkter Zugriff auf die ersten NZE einer Spalte möglich ist. Bei **KS-3A** (sowie bei **DE-2A** und **BP-2A**) muss die Datenstruktur linear durchsucht werden. Bei **2L-2DL** sind die NZE zwar spaltenweise verkettet, das erste NZE jeder Spalte muss jedoch zuerst durch Suchen der entsprechenden Spalte in der `colList` gefunden werden. Bei **GS-3A**, **1A-1DL**, **1A-ZA** und **1L-1DL** muss zeilenweise gesucht werden, bis das erste NZE in der entsprechenden Spalte gefunden wird.
2A-2DL bleibt mit steigender Dimension fast konstant, nur *Cache-Misses* erhöhen die Zugriffszeit ein wenig.
Die Zeit des **Dense**-Schemas nimmt zu, da – wie bei `GetNextInRow` – mit zunehmender Dimension immer mehr Null-Elemente duchsucht werden müssen.

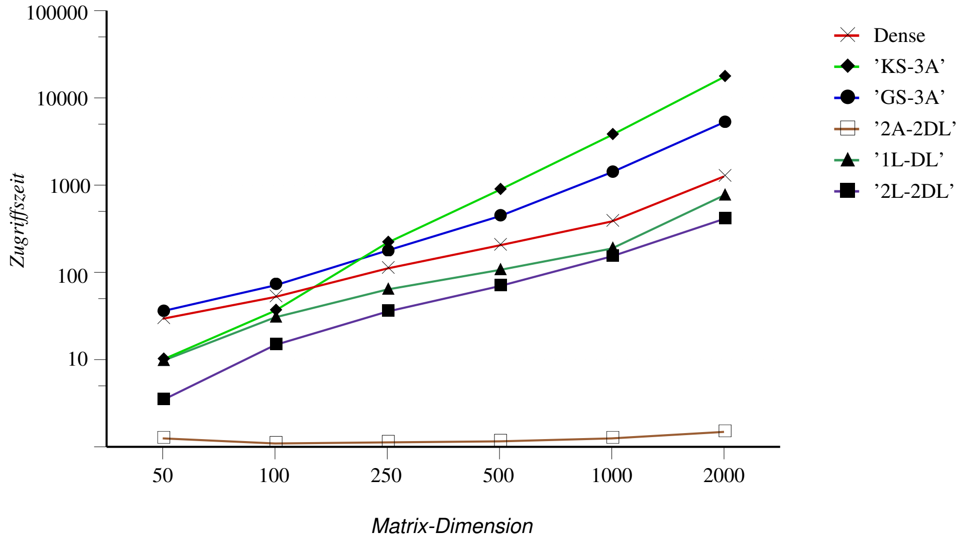


Abbildung 6. Zugriffszeiten `GetNextInCol - Reset=1`

Zugriff auf weitere NZE einer Spalte: `GetNextInCol - Reset=0`

Die Zeiten für **2A-2DL** und **2L-2DL** bleiben unabhängig von der Dimension konstant, da ein direkter Zugriff auf das nächste NZE in derselben Spalte möglich ist.
KS-3A, **BP-2A** und **DE-2A** nehmen stark zu, da das nächste NZE nur linear gesucht werden kann. Bei den Schemata **1A-1DL** und **1L-1DL** ist ebenfalls nur lineares Durchsuchen der Zeilen möglich. Bei **GS-3A** und **1A-ZA** ist der Anstieg weniger stark, da innerhalb der Zeilen binär gesucht werden kann.
Die Zugriffszeit für das **Dense**-Schema bleibt mit zunehmender Dimension ungefähr konstant.

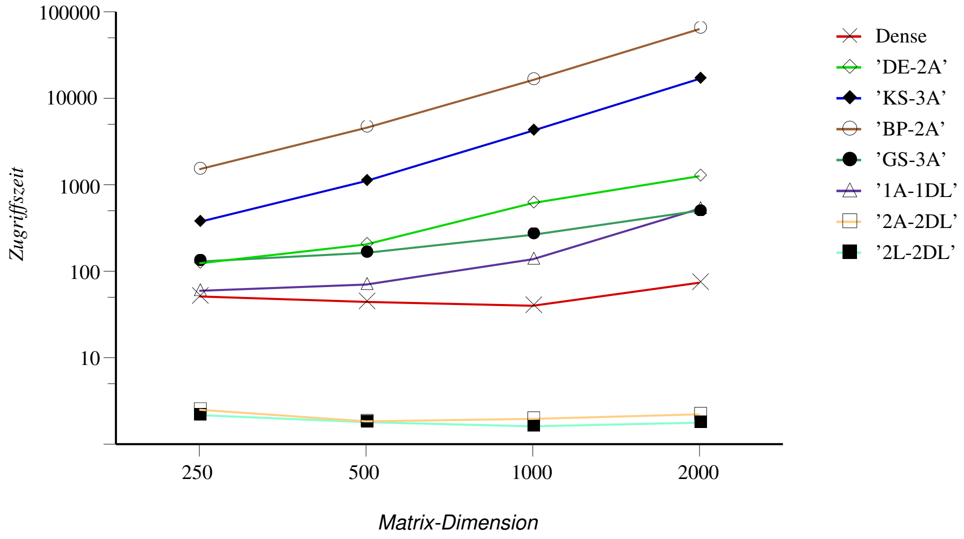


Abbildung 7. Zugriffszeiten `GetNextInCol - Reset=0`

3.2.2. Testserie 2: Variation der Besetztheit

Lesezugriff auf einzelne Elemente: `GetElem`

Die Zugriffzeiten des **DE-2A** Schemas erhöhen sich sehr stark mit der Zahl der NZE, da dessen Datenstruktur nur linear durchsucht werden kann. Die Zeiten von **KS-3A**, **BP-2A** und **GS-3A** erhöhen sich dem gegenüber weniger stark, da binäres Suchen verwendet wird, wobei bei **GS-3A** nur jeweils eine Zeile durchsucht werden muss. **1A-ZA** verhält sich fast gleich wie **GS-3A**.
Die Zunahme der Zeiten der Schemata mit verketteten Listen (**1A-1DL**, **2A-2DL**, **1L-1DL** und **2L-2DL**) mit zunehmender Anzahl NZE lässt sich ebenfalls dadurch erklären, dass die Listen der Elemente nur sequentiell durchsucht werden können. Bei **2L-2DL** und **1L-1DL** kommt zusätzlich die Zeit hinzu, die zum Finden der gewünschten Zeile bzw. Spalte in der `rowList` bzw. `colList` benötigt wird. Diese Zeit ist relativ gross, aber unabhängig von der Anzahl NZE.
Die Zugriffszeit beim **Dense**-Schema ist konstant, da direkt auf jedes Element zugegriffen werden kann. Ab einer gewissen Besetztheit ist diese Speichermethode die effizienteste beim Zugriff auf einzelne Elemente.

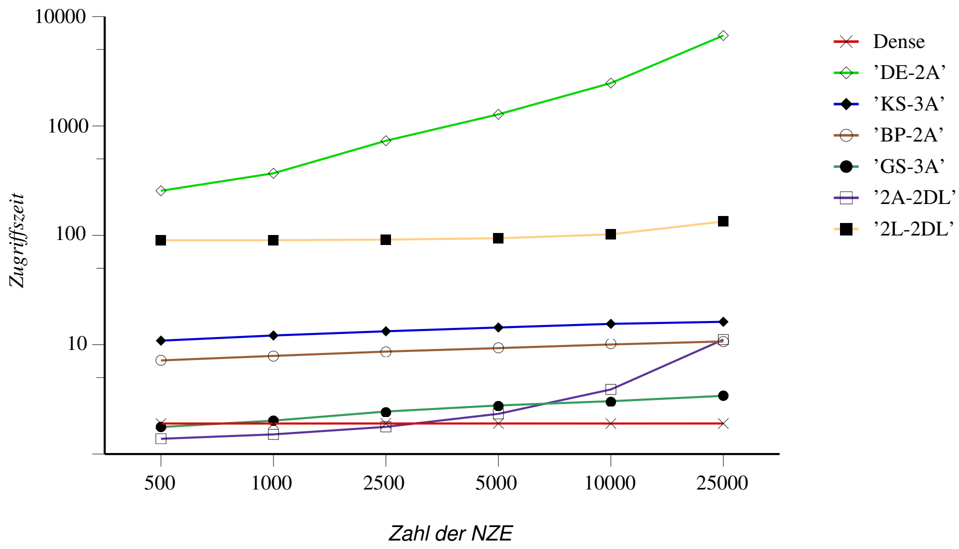


Abbildung 8. Zugriffszeiten `GetElem`

Schreibzugriff auf einzelne Elemente: `SetElem`

Bei den Schemata **BP-2A**, **DE-2A** und **KS-3A** erhöht sich die Zugriffszeit mit der Anzahl gespeicherter NZE, da ihre Datenstrukturen (bei diesem Test) nicht sortiert waren und daher linear durchsucht werden mussten.
Beim **GS-3A** Schema nimmt die Zeit mit grösserer Zahl NZE zuerst ab, später jedoch wieder zu. Der Grund ist, dass der Initialaufwand (Zuweisung der Suchindizes) im Vergleich zum Suchen (Pointererhöhung) relativ gross ist und bei kleiner Zahl von NZE pro Zeile den grössten Anteil der Zugriffszeit beansprucht. Bei vielen NZE pro Zeile nimmt dann die Suchzeit überhand. Ähnliches gilt für das **2A-2DL** Schema.
Die Zeit beim **Dense**-Schema verringert sich mit grösserer Zahl von NZE, da die Datenlokalität öfters ausgenutzt wird (weniger *Cache-Misses*).

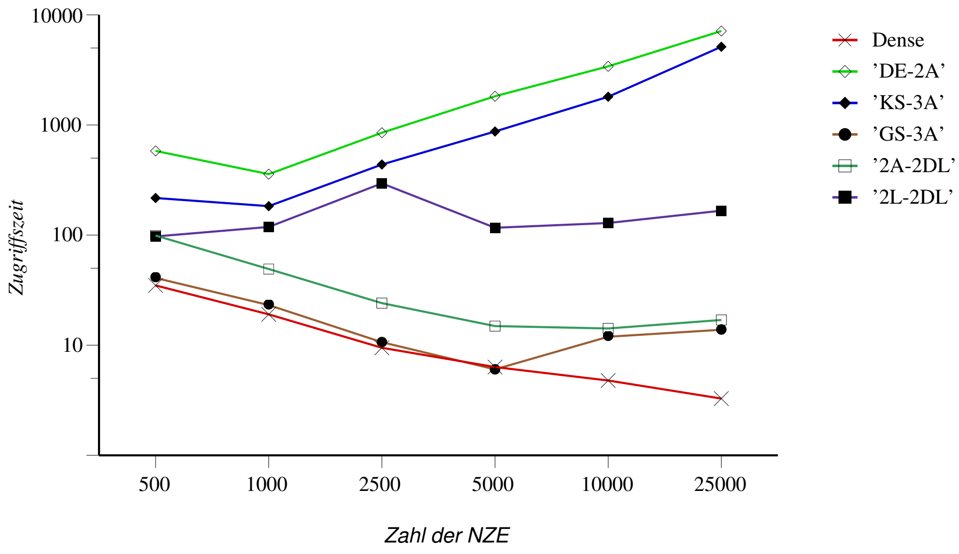


Abbildung 9. Zugriffszeiten `SetElem`

Einfügen von NZE: `SetElem - Insert`

Die gemessenen Zugriffszeiten beim Einfügen in die Array-Schemata (**KS-3A**, **BP-2A**, **GS-3A** und **DE-2A**) hängen stark von der implementierten Einfügeart ab. Beim **DE-2A**-Schema muss zuerst die Stelle des Einfügens linear gesucht werden und anschliessend alle Elemente im Array hinter dieser Position um eins nach hinten geschoben werden, um Platz für das neue NZE zu schaffen. Die Zunahme beim **KS-3A** Schema kommt daher, dass das neue NZE am Schluss angefügt wird, beim nächsten Einfügen jedoch nicht mehr binäres Suchen angewendet wird (beim Prüfen, ob ein NZE schon gespeichert ist).
Die Zeiten beim **GS-3A**-Schema bleiben bei diesem Test gleich, weil die Elemente zeilenweise eingefügt werden, so dass die neuen Elemente immer am Schluss der Datenstruktur stehen, und somit keine Elemente nach hinten geschoben werden müssen.
Bei sehr wenigen NZE pro Zeile ist das **2A-2DL** wie auch das **1A-1DL** Schema etwas langsamer als bei mehreren NZE, da das Einfügen des ersten Elementes in eine Liste einen etwas grösseren Aufwand verursacht als das Einfügen weiterer Elemente. Bei noch grösserer Zahl NZE nimmt dann die Suchzeit innerhalb der Listen zu, was zu insgesamt langsameren Zugriffszeiten führt (ähnliches gilt für die **1L-1DL** und **2L-2DL** Schemata).
Die starke Abnahme beim **Dense**-Schema mit zunehmender Zahl NZE lässt sich dadurch erklären, dass wiederum die Datenlokalität der eingefügten Elemente zu weniger *Cache-Misses* führt (die Daten im Cache müssen weniger oft in den Hauptspeicher ausgelagert werden).

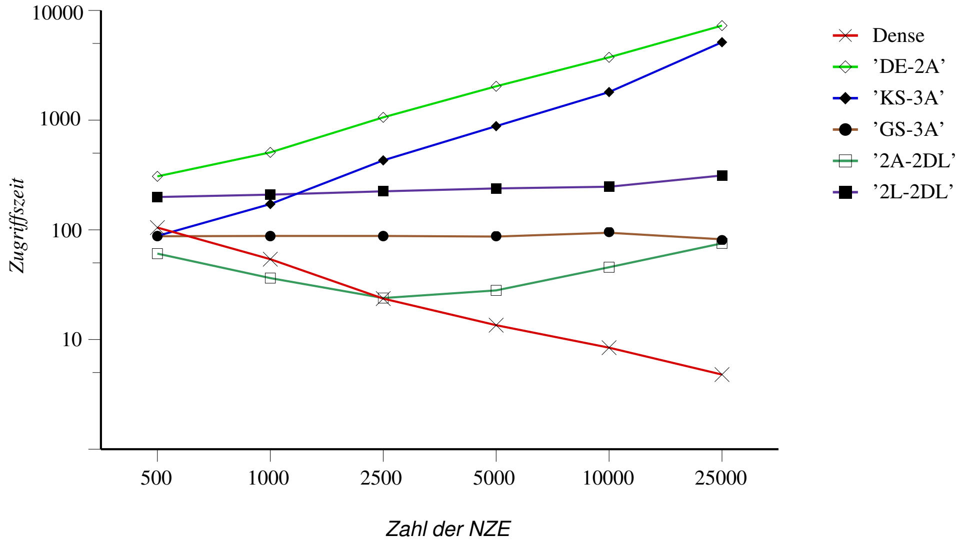


Abbildung 10. Zugriffszeiten `SetElem - Insert`

Zugriff auf die ersten NZE einer Zeile: `GetNextInRow - Reset=1`

Alle Schemata mit direktem Zugriff auf die ersten gespeicherten Elemente in einer Zeile (**GS-3A**, **1A-ZA**, **1A-1DL**, **2A-2DL**, **1L-1DL** und **2L-2DL**) sind sehr schnell und die Zeiten bleiben für zunehmende Besetztheit konstant. Beim **1L-1DL** und **2L-2DL** Schema kommt jedoch noch die Zeit für die Suche der entsprechenden Zeile (in `RowList`) hinzu (diese Zeit ist unabhängig von der Zahl der NZE).
Beim **KS-3A** und **BP-2A** Schema nehmen die Zeiten zu, weil die Datenstruktur (binär) durchsucht werden muss. Die starke Zunahme beim **DE-2A** erklärt sich dadurch, dass nur linear gesucht werden kann.
Die Zeiten des **Dense**-Schemas nehmen leicht ab, da weniger Null-Elemente durchsucht werden müssen, bis das erste NZE gefunden ist.

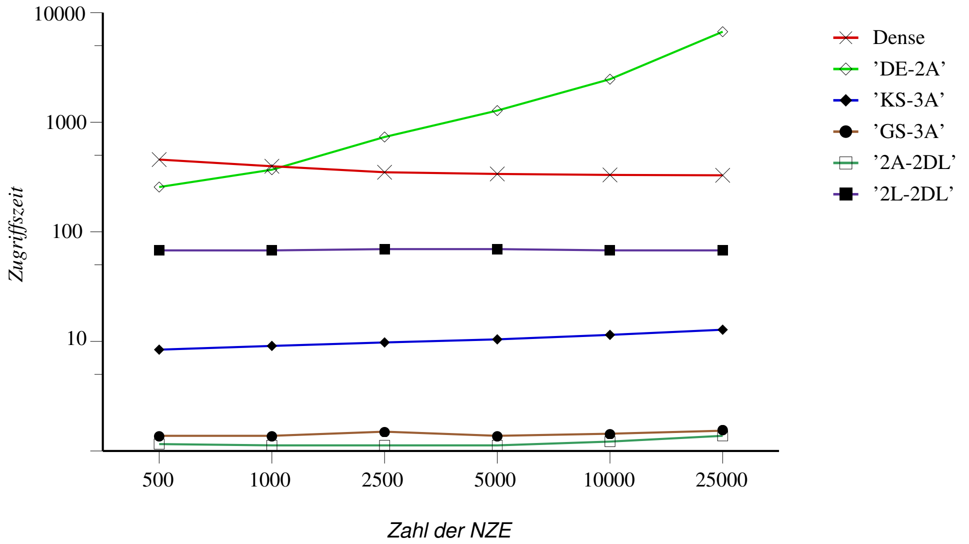


Abbildung 11. Zugriffszeiten `GetNextInRow - Reset=1`

Zugriff auf weitere NZE einer Zeile: `GetNextInRow - Reset=0`

Alle Schemata sind bei diesem Test sehr schnell. Da die Datenstrukturen bei den Array-Schemata zeilenweise sortiert wurden, bestand der Zugriff auf das nächste NZE in derselben Zeile meist nur aus einer Erhöhung einer Variablen und einigen Vergleichen (z.B. Index-Check). Die etwas grösseren Zeiten bei der Matrix mit 1000 NZE stammen vom Testprogramm (Initialisierung einer While-Schleife, die nur einmal durchlaufen wird).
Die leichte Zunahme der Zeiten für stärker besetzte Matrizen bei den Schemata mit dynamischen Listen lässt sich wiederum auf *Cache-Misses* zurückführen, d.h. beim Zugriff auf das nächste NZE muss öfters ein neuer Speicherbereich aus dem Hauptspeicher nachgeladen werden.
Die Abnahme beim **Dense**-Schema ist wieder dadurch zu erklären, dass weniger Null-Elemente durchsucht werden müssen.

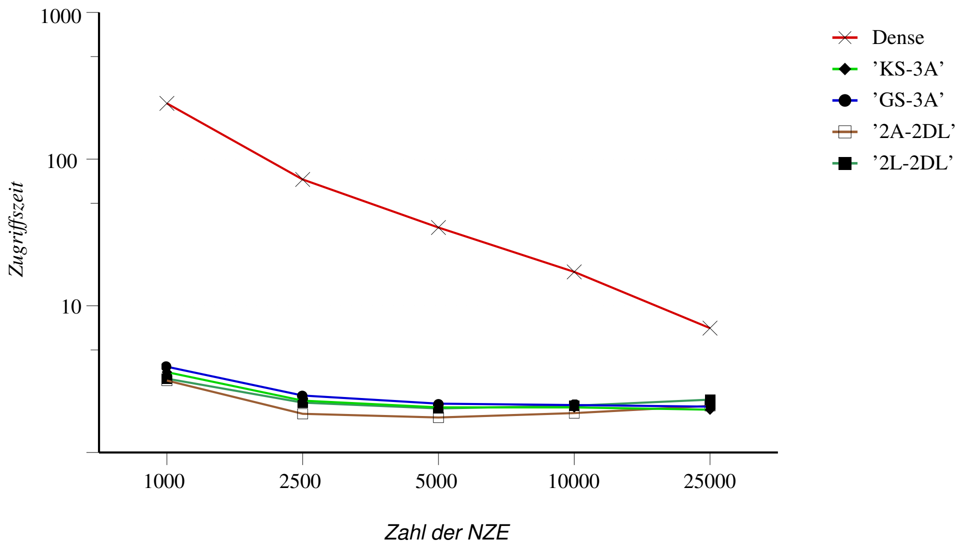


Abbildung 12. Zugriffszeiten `GetNextInRow - Reset=0`

Zugriff auf die ersten NZE einer Spalte: `GetNextInCol - Reset=1`

Die Zugriffszeiten bei doppelt verketteten Listen (**2A-2DL** und **2L-2DL**) bleiben konstant, unabhängig von der Anzahl der gespeicherten NZE. Der Aufwand bei **2L-2DL** zum Auffinden der gewünschten Spalte in der `colList` ist jedoch – verglichen mit dem direkten Spalten-Zugriff bei **2A-2DL** – sehr hoch.
Bei allen anderen Schemata erhöhen sich die Zugriffszeiten stark, da jeweils zeilenweise gesucht werden muss, bis das erste NZE in der gewünschten Spalte gefunden ist. Der Aufwand dieses zeilenweisen Suchens erhöht sich mit der Zahl der gespeicherten NZE pro Zeile, es müssen jedoch weniger Zeilen durchsucht werden, da das erste NZE in einer Spalte durchschnittlich einen kleineren Zeilen-Index hat. **GS-3A** ist das einzige Schema, bei dem binäres Suchen angewendet werden kann.
Die Zeiten beim **Dense**-Schema verringern sich, da wiederum weniger Null-Elemente durchsucht werden müssen, wenn sich die Zahl der NZE pro Spalte erhöht.

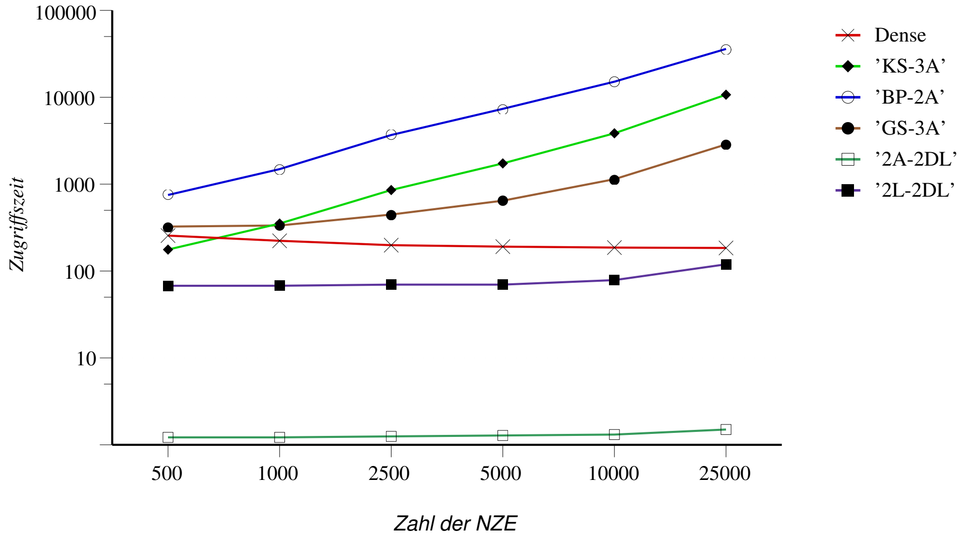


Abbildung 13. Zugriffszeiten `GetNextInCol - Reset=1`

Zugriff auf weitere NZE einer Spalte: `GetNextInCol - Reset=0`

Auch bei diesem Test haben die Schemata mit doppelt verketteten Listen einen grossen Vorteil, ihre Zugriffszeiten sind viel kürzer als bei den Array-Schemata oder den Schemata mit einfach verketteten Listen. Die Zeiten von **KS-3A** und **BP-2A** erhöhen sich wieder sehr stark (lineares Suchen).
Bei Schemata mit direktem Zugriff auf eine Zeile (**GS-3A**, **1A-1DL** und **1L-1DL**) nehmen die Zeiten mit zunehmender Besetztheit langsam ab, da weniger Zeilen durchsucht werden müssen, bis ein NZE in der gewünschten Spalte gefunden wird (obwohl mehr Elemente durchsucht werden müssen). Wird die Matrix jedoch noch stärker besetzt, so führt das lineare Durchsuchen der verketteten Listen (**1A-1DL** und **1L-1DL**) wieder zu einem Anstieg der Zugriffszeit.
Auch die Zeiten des **Dense**-Schemas nehmen – ähnlich wie beim zeilenweisen Zugriff – wieder stark ab. Der spaltenweise Zugriff ist dabei schneller als der zeilenweise.

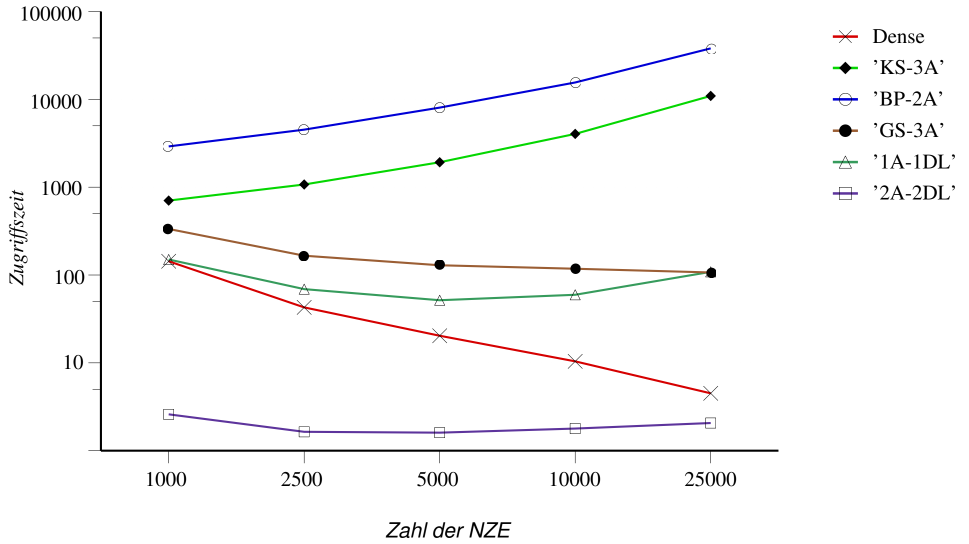


Abbildung 14. Zugriffszeiten `GetNextInCol - Reset=0`

3.3. Beurteilung

- **KS-3A**
  Dieses Schema ist recht langsam, weil keine Möglichkeit besteht, direkt auf die Elemente einer Zeile bzw. Spalte zuzugreifen, es muss jeweils die gesamte Datenstruktur durchsucht werden. Ist diese sortiert, kann die Suchzeit beträchtlich verringert werden ($log_2(r)$ anstatt $r/2$ Vergleiche). Spaltenweiser Zugriff ist nur dann schnell, wenn die Datenstruktur spaltenweise sortiert ist, dann ist aber der zeilenweise Zugriff langsam.
  Der Speicher-Overhead ist konstant 100%. Durch Ersetzen dieses Schemas mit dem **BP-2A** Schema – welches fast die gleichen Zugriffszeiten aufweist – lässt sich der Speicher-Overhead auf 50% reduzieren.

- **BP-2A**
  Dieses Schema entspricht dem **KS-3A** Schema, mit dem Vorteil des kleineren Speicher-Overheads. Der Nachteil ist jedoch, dass der spaltenweise Zugriff noch langsamer ist als beim **KS-3A** Schema, weil mehr Berechnungen in den Vergleichen durchgeführt werden müssen.

- **GS-3A**
  Dies ist das schnellste Schema mit Arrays und wird deshalb oft in Software-Paketen verwendet. Der Speicher-Overhead ist klein und nur der spaltenweise Zugriff ist relativ langsam. Dies lässt sich jedoch umgehen, wenn Algorithmen verwendet werden, die nur zeilenweisen Zugriff benötigen oder andernfalls das Schema spaltenweise aufgebaut wird (mit einem Array `colPtr`, das auf das jeweils erste Element einer Spalte zeigt).

- **DE-2A**
  Dies ist mit Abstand das langsamste Schema. Weder kann ein direkter Zugriff auf die ersten Elemente einer Zeile erfolgen (wie z.B. bei **GS-3A**), noch kann ein schnelles Suchverfahren (binäres Suchen) angewendet werden.

- **2A-2DL**
  Dies ist das schnellste aller verglichenen Schemata, der zeilen- und spaltenweise Zugriff ist dank der zweidimensional verketteten Listen gleich schnell. Leider hat es einen grossen Speicher-Overhead.

- **2L-2DL**
  Langsamer als **2A-2DL** hat dieses Schema zudem einen grösseren Speicher-Overhead. Der Vorteil dieses Schemas gegenüber **2A-2DL** ist, dass dynamisch ganze Zeilen und Spalten einfügt und gelöscht werden können. Bei grossen Matrizen wird jedoch die Suchzeit in der `rowList` bzw. der `colList` sehr gross.

- **1A-1DL**
  Auch dieses Schema ist sehr schnell, mit Ausnahme des spaltenweisen Zugriffs (weil die Elemente nicht spaltenweise verkettet sind und somit meist mehrere Zeilen-Listen durchsucht werden müssen). Ist kein spaltenweiser Zugriff erforderlich, eignet sich dieses Schema besser als das **2A-2DL** Schema, weil der Speicher-Overhead kleiner ist. Es entspricht praktisch einer verketteten Version des **GS-3A** Schemas und hat auch vergleichbare Zugriffszeiten.

- **1L-1DL**
  Dies ist die eindimensionale Variante des **2L-2DL** Schemas und hat dieselben Nachteile. Auch hier sind der Speicher-Overhead und die Zugriffszeiten grösser als beim **1A-1DL** Schema. Der Vorteil ist wiederum, dass dynamisch zur Laufzeit neue Zeilen eingefügt und nicht mehr gebrauchte gelöscht werden können.

- **1A-ZA**
  Dieses Schema entspricht dem **GS-3A** Schema, wobei das Array `rowPtr` durch `rowArray` ersetzt wurde. Der Speicher-Overhead ist ähnlich gross und die Zugriffszeiten sind fast gleich. Beim Einfügen ist dieses Schema jedoch das schnellste für grosse Matrizen.
  Der Vorteil gegenüber dem **GS-3A** Schema besteht darin, dass beliebig viele Elemente alloziert werden können, indem ein volles Daten-Array neu und grösser alloziert wird und die bestehenden Elemente dieser Zeile umkopiert werden.

3.4. Schlussfolgerungen

Ein bestes, für alle Operationen und Algorithmen geeignetes Speicher-Schema gibt es nicht. Es muss jeweils ein Kompromiss zwischen Speicher-Overhead und Zugriffszeiten eingegangen werden. Die Wahl des geeigneten Schemas hängt von mehreren Faktoren ab:

- Struktur der Matrix (Band-Matrix, symmetrische Matrix, Block-Matrix).
- Operationen und Algorithmen die auf der Matrix ausgeführt werden (Multiplikation, Faktorisierung, etc.).
- Zielarchitektur (Parallel- oder Vektor-Rechner, RISC-Workstation).
- Programmiersprache (Zeiger und Allozieren zur Laufzeit möglich oder nicht).

Oft ist es auch sinnvoll, während eines Algorithmus (oder eines Teiles davon) ein Schema in ein anderes, besser geeignetes umzukopieren bzw. die Resultate in ein Schema zu speichern, das für den nächsten Schritt geeignet ist. Durch geschickte Wahl des Schemas kann die Laufzeit eines Algorithmus beträchtlich verkürzt werden.

Zum Schluss – kurz zusammengefasst – eine Übersicht, welches der hier vorgestellten Schemata sich in welchen Situationen eignet:

Ist der allozierte Speicher ein beschränkender Faktor und wird zur Laufzeit keine grosse Zahl von neuen NZE erzeugt, so wird am besten das **GS-3A** Schema benutzt. Es wird in vielen Software-Paketen verwendet und in mehreren Büchern vorgestellt. Viele Algorithmen wurden auf dieses Schema optimiert.
Sollen zur Laufzeit beliebig viele NZE eingefügt werden können, so wird vorzugsweise das **1A-1DL** oder das **1A-ZA** Schema verwendet. Die Zugriffszeiten sind schnell (wobei **1A-ZA** durchschnittlich schneller ist als **1A-1DL**) und der Speicher-Overhead ist nicht sehr gross.
Soll der zeilen- und spaltenweise Zugriff schnell sein, so wird das **A2-2DL** Schema eingesetzt. Der Nachteil ist dann der grössere Speicheroverhead.
2L-2DL und **1L-1DL** eignen sich nur, falls sich zur Laufzeit die Grösse der Matrix ändert bzw. Zeilen eingefügt und gelöscht werden müssen. Diese Flexibilität des Einfügens wird jedoch – vor allem bei grossen Matrizen – mit erheblich langsameren Zugriffszeiten erkauft.
KS-3A eignet sich wegen seiner Einfachheit vor allem als Dateiformat zum Speichern einer Sparse-Matrix (siehe z.B. MATLAB [17]), nicht jedoch als Repräsentation bei Operationen auf der Matrix.

Literaturverzeichnis

[1] Fernando L. Alvarado *The Sparse Matrix Manipulation System, User and Reference Manual*, The University of Wisconsin, Madison, 1993.
Available via anonymous ftp: `ftp://eceserv0.ece.wisc.edu/pub/smms93/`

[2] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J.J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov and D. Sorensen, *LAPACK User's Guide, Second Edition*, SIAM, Philadelphia, 1995.

[3] S. Carney, M.A. Heroux, G. Li, R. Pozo, K.A. Remington and K. Wu, *A Revised Proposal for a Sparse BLAS Toolkit*, SPARKER Working Note #3, 1996

[4] J.J. Dongarra, C.B. Moler, J.R. Bunch, G.W. Stewart, *LINPACK User's Guide*, SIAM, Philadelphia, 1979.

[5] J.J. Dongarra, E. Grosse, *Distribution of mathematical software via electronic mail*, In Communications of the ACM, 30(5), pp. 403–407, July 1987.

[6] J.J. Dongarra, J. Du Croz, S. Hammarling and R. Hanson, *An Extended Set of Fortran Basic Linear Algebra Subprograms*, ACM Trans. Math. Software 14, 1988.

[7] J.J. Dongarra, J. Du Croz, I.S. Duff and S. Hammarling, *A Set of Level 3 Basic Linear Algebra Subprograms*, ACM Trans. Math. Software 16, 1990.

[8] J.J. Dongarra, A. Lumsdaine, R. Pozo and K.A. Remington, *A Sparse Matrix Library in C++ for High Performance Architectures*, Proceddings of the Second Annual Object-Oriented Numerics Conference, 1994, pp. 214 - 218.

[9] Iain S. Duff, A.M. Erisman and J.K. Reid, *Direct Methods for Sparse Matrices*, Oxford University Press, 1986

[10] Iain S. Duff, *User's Guide for the Harwell-Boeing Sparse Matrix Collection*, CERFACS, 42 Ave G. Coriolis, 31057 Toulouse Cedex, France

[11] Chenyi Hu, Anna Frolov, R. Baker Kearfott and Quing Yang, *A general iterative sparse linear solver and its parallelization for interval Newton methods*, Reliable Computing 1 (3) (1995), pp. 251–263

[12] Alan Jennings and J.J. McKeown, *Matrix Computation, Second Edition*, John Wiley & Sons, Chichester, 1992.

[13] Kenneth S. Kundert, Alberto Sangiovanni-Vincentelli, *Sparse - Version 1.3, A Sparse Linear Equation Solver*, Department of Electrical Engineering and Computer Science, University of California, Berkeley, 1988.
Distributed by `netlib (http://www.netlib.org)`.

[14] Sergio Pissanetsky, *Sparse Matrix Technologie*, Academic Press, London, 1984

[15] R. Pozo, Karin A. Remington and A. Lumsdaine, *SparseLib++ v. 1.5, Sparse Matrix Class Library, Reference Guide*, University of Notre Dame, 1996.
SparseLib++ Homepage: `http://gams.nist.gov/acmd/Staff/RPozo/sparselib++.html`

[16] Youcef Saad, *Sparskit - A Basic Tool Kit For Sparse Matrix Computations*, Computer Science Department, University of Minnesota, Minneapolis, 1994.
Available via `http:www.cs.umn.edu/Research/arpa/SPARSKIT/sparskit2.html` or `ftp: ftp.cs.umn.edu`

[17] Kermit Sigmon, *MATLAB Primer - Third Edition*, Department of Mathematics, University of Florida, Gainesville, 1993

[18] David E. Steward and Zbigniew Leyk, *Meschach Library - Version 1.2b*, School of Mathematical Sciences, Australian National University, Canberra, 1994.
Distributed by `netlib (http://www.netlib.org)`.

[19] M. Veldhorst, *An Analysis of Sparse Matrix Storage Schemes*, Mathematical Centre Tracts 150, Mathematisch Centrum, Amsterdam, 1982.

[20] Visual Numerics Inc., *IMSL - Graphical and Numerical Libraries*, Visual Numerics Inc., 1996, `http://www.vni.com`