|  |
| --- |
| FH-Aachen |
| Gauß-Verfahren für dünn besetzte Matrizen |
| COBOL |

|  |
| --- |
| Björn Lüpschen & Philipp Kohl  [Wählen Sie das Datum aus] |

Inhaltsverzeichnis

[Möglichkeiten zur Abspeicherung dünn besetzter Matrizen (n x m) 3](#_Toc472799228)

[Standard Speicherart: 3](#_Toc472799229)

[Compressed Row Storage (CRS)/Compressed Column Storage (CCS) 3](#_Toc472799230)

[Diskussion der Speicherformen 4](#_Toc472799231)

[Zu speichernde Elemente 4](#_Toc472799232)

[Verwaltungsaufwand 5](#_Toc472799233)

[Fazit 6](#_Toc472799234)

# Möglichkeiten zur Abspeicherung dünn besetzter Matrizen (n x m)

## Standard Speicherart:

Die komplette Matrix wird mit all ihren Elementen gespeichert.

|  |  |
| --- | --- |
| **Vorteile** | **Nachteile** |
| Einfache Handhabung, da man direkt auf die einzelnen Elemente der Matrix zugreifen kann | n x m Elemente müssen gespeichert werden, obwohl 70% der Einträge 0 sind -> großer Speicherverbrauch |
| Höhere Performance der Algorithmen, da das Decodieren entfällt |  |

## Compressed Row Storage (CRS)/Compressed Column Storage (CCS)

Eine alternative Speicherform, um dünnbesetzte Matrizen zu speichern ist die Compressed Row Storage (CRS) oder Compressed Column Storage (CCS).   
Die Vorgehensweise soll anhand des CRS-Verfahrens erläutert werden:  
Gegeben sei eine Matrix (Indizes starten bei 0!):  
  
  
Es werden die Werte, die ungleich 0 sind, in einem Array **Werte** gespeichert. Dabei wird die Reihenfolge berücksichtigt. Zeilenweise von links nach rechts.  
  
  
  
Zu jedem Wert im Array **Werte** wird in einem zweiten Array **Spalten-Index** der korrespondierende Spalten-Index hinterlegt. Das heißt, dass die Arrays **Werte** und **Spalten-Index** immer die gleiche Elementanzahl aufweisen.

In einem dritten Array **Zeilen-Pointer** wird die Anzahl der Elemente pro Zeile codiert:  
Der erste Wert ist immer die 0. Die weiteren Einträge sind die kumulierten Anzahlen der Nicht-null-Elemente pro Zeile. Somit ist der letzte Wert immer die Anzahl aller Nicht-null-Elemente der Matrix. Die Einträge dieses Arrays zeigen die Grenzen einer Zeile an.   
  
  
  
Aus benachbarten Werten kann man auf die Indizes für **Spalten-Index** und **Werte** schließen, um eine Zeile zu rekonstruieren.  
Ein Beispiel: 2. Zeile rekonstruieren  
Um die zweite Zeile zu rekonstruieren, benötigt man das zweite „2-er-Tupel“.  
  
  
  
Die **3** gibt uns nun den Start-Index und die **5** den End-Index für die Arrays **Spalten-Index** und **Werte** (sowie die Anzahl der Elemente in dieser Zeile (5-3=2)):

Nun hat man alle Informationen, die man braucht: Zeilennummer, Spaltennummer und die dazugehörigen Werte. Anhand der 0 in **Spalten-Index** und4 in **Werte** weiß man, dass die 4 in der ersten Spalte stehen muss. Analog dazu kann man ablesen, dass die 2 in der dritten Spalte steht. Die restlichen Elemente der zweiten Zeile mit Nullen auffüllen.

Die Compressed Column Storage läuft analog zum CRS-Verfahren, nur, dass als Betrachtungsbereich nicht die Zeilen sondern die Spalten gelten. Sprich: Werte werden Spaltenweise einsortiert, es gibt ein Array um die Zeilen-Indizes zu speichern und ein Spalten-Pointer-Array.

|  |  |
| --- | --- |
| **Vorteile** | **Nachteile** |
| Weniger Speicherverbrauch | Zeilen/Spalten müssen vor Verwendung wiederhergestellt werden |
| Schnelle Suche von Matrixelementen | Hinzukommen von Nicht-Null-Elementen und Entfernen der Null-Elemente erfordern ständiges umordnen -> geringe Performance |

# Diskussion der Speicherformen

## Zu speichernde Elemente

1. Standard Speicherart:  
   Bei der Standard Speicherart müssen alle Elemente abgespeichert werden: n x m  
   Beispiel: 10x10 = 100
2. Compressed Row Storage:  
   Worst-Case (genau 30% der Elemente sind ungleich null):   
   Für Werte und Spalten-Index: 2 x (0.3 x n x m) = 0.6 x n x m  
   Für Zeilen-Pointer: n + 1  
   Insgesamt: 0.6 x n x m + (n + 1)  
   Beispiel: 10x10 = 71

## Verwaltungsaufwand

1. Standard Speicherart:  
   Bei der Standard Speicherart ist keine weitere Verwaltung nötig. Man kann über die Indizes direkt auf die einzelnen Elemente zugreifen.
2. Compressed Row Storage:

Beim CRS-Verfahren ist eine weitere Verwaltung nötig, da sich durch das Hinzukommen oder Wegfallen eines Nicht-null-Elements die Speicher-Arrays ändern.  
Fällt ein Wert aus der Matrix weg (wird 0), muss dieser Eintrag aus den beiden Arrays **Werte** und **Spalten-Index** entfernt werden. Daraus folgt unter Umständen eine Neuordnung des gesamten Arrays, sofern nicht das letzte Nicht-null-Element gelöscht wurde.  
  
Gegeben sei folgende Matrix:

Daraus folgen die Verwaltungs-Arrays:

Würde die in der zweiten Zeile, dritten Spalte wegfallen, müssen die rot markierten Elemente in **Werte** und **Spalten-Index** entfernt werden.

Gefolgt davon müssen alle nachfolgenden Elemente nach vorne gezogen werden.

In **Zeilen-Pointer** muss der End-Index der zweiten Zeile bzw. Start-Index der dritten Zeile geändert werden. In diesem Fall von 5 auf 4, da die zweite Zeile nur noch einen Wert enthält, der nicht Null ist. Auch wenn in einer Zeile nur Nullen stehen würden, hat dieses Array immer die gleiche Länge. Es wird nichts gelöscht (Es würde nur zweimal die gleiche Zahl im Array hintereinander stehen.)  
  
Analog dazu wird mit dem Hinzufügen eines Elementes ungleich Null verfahren, nur, dass keine Elemente entfernt, sondern eingefügt werden.  
  
Fazit:

Pro Entfernung eines Wertes müssen 2 Einträge gelöscht und einer geändert, sowie gegebenenfalls die Arrays defragmentiert werden.

Pro Hinzufügen eines Wertes müssen 2 Einträge hinzugefügt und einer geändert, sowie gegebenenfalls die Arrays defragmentieren werden.

## Fazit

Das CRS-Verfahren bietet Vorteile in der Menge des benötigten Speichers, hat aber Nachteile in der Verwaltung und Performance.  
Die Standard Speicherart hat Vorteile in der Verwaltung aber einen Nachteil in der Speichermenge.  
Da man in Cobol auf dynamische Speicherallokierung verzichten muss, muss man immer vom Worst-Case ausgehen. Auch wenn es in Cobol die Möglichkeit gibt mit „DEPENDING ON“ nicht benötigten Speicher wieder freizugeben, muss ebenfalls beachtet werden, dass auch wieder Elemente hinzugefügt werden. Dabei muss die Obergrenze immer fest gesetzt werden.  
Somit kommt man immer noch auf gut 70% des Speicherbedarfs (30% Ersparnis).  
Für etwa 30% Speicherplatzersparnis muss man die Verwaltungen und Performance Einbuße in Kauf nehmen. Deshalb haben wir uns an dieser Stelle für die Standard Speicherart entschieden.  
  
Array, ca 70%, Verwaltung, COBOL muss man immer eine feste obergrenze angeben