Wettbewerb:	**Jugend forscht**
Teilnahmejahr:	2003
Bundesland:	Nordrhein Westfalen
Fachgebiet:	Mathematik / Informatik
Online Anmeldung Nr:	1086

Titel der Arbeit: **T**opdown **V**orverarbeitendes **D**eterminanten **S**chema (**TVDS**): Optimierung des Laplace'schen Entwicklungsschemas für Determinanten

Einzelteilnehmer

Geschlecht:	männlich
Name:	Engelmann
Vorname:	Viktor
Straße:	Leyboldstr. 1
PLZ/Ort:	50968 Köln
Telefon:	0221 / 3796855
Geb.-Dat.:	07.09.1981
Schule:	Gymnasium der Stadt Kerpen / Europaschule
Straße:	Philipp-Schneider-Str.
PLZ/Ort:	50171 Kerpen
Telefon:	02237 / 3041
Betreuungslehrer:	Helmut Schumacher
Klassenstufe:	13

Alle hier gemachten Angaben sind verbindlich und dürfen für Presse- und Öffentlichkeitsarbeit der Stiftung Jugend forscht verwendet werden. Außerdem versichere ich, alle benutzten Quellen angegeben zu haben.

Mit meiner Unterschrift erkenne ich die Teilnahmebedingungen des Wettbewerbs Jugend forscht an.

_____	_____	_____
Datum	**Ort**	**Unterschrift**

Inhalt

Inhalt 1
Die Determinante 2
Das Laplace'sche Entwicklungsschema 2
Beschreibung des TVDS 3
Der TVDS-Baum 5
Leistungsanalyse anhand einer Aufwandsabschätzung 6
Beispielaufgabe 8
Ersparnisdemonstration 9
Quellcode (in c++) 10
Implementierung des TVDS-Baumes. 11
Testläufe 12
Leistungsanalyse anhand von Zeitmessungen 13
Aussicht 14
Danksagungen 14
Bild- und Literaturnachweis 14

Die Determinante

Jede quadratische Matrix hat eine sog. „Determinante“. Diese Determinante ist in der Linearen Algebra von Bedeutung im Zusammenhang mit der Matrizeninversion (eine Matrix, die die Determinante 0 hat ist nicht invertierbar) und besonders zur Lösung homogener linearer Gleichungssysteme, wodurch man sie z.B. bei affinen Abbildungen zur Bestimmung von Eigenwerten einsetzt (=> eine Abbildungsmatrix A bildet Objekte auf Objekte mit det(A)-fachem Flächeninhalt ab). Außerhalb der Linearen Algebra ist sie z.B. von Bedeutung für die Integrationstheorie für Funktionen mehrerer Variablen, weil sie eng mit dem Begriff des Volumens zusammenhängt. Diese Arbeit setzt sich aber nicht mit den Anwendungsgebieten der Determinante auseinander, sondern mit ihrer Berechnung.

Das Laplace’sche Entwicklungsschema

Die Determinante einer 2x2 Matrix wird folgendermaßen berechnet:

$$\det\begin{pmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{pmatrix} = a_{1,1} * a_{2,2} - a_{2,1} * a_{1,2}$$

Hauptdiagonale - Nebendiagonale

Die Determinante einer nxn Matrix kann rekursiv berechnet werden: man wählt eine Zeile z, durchläuft alle Spalten s, berechnet die Unterdeterminante $\det(A_{z,s})$ (Die Determinante der Matrix ohne die Zeile z und die Spalte s, man sagt: „man streicht die Zeile/Spalte“ bzw. „man entwickelt nach der Zeile z und der Spalte s“) und multipliziert sie mit $a_{z,s}$,dem Element der Matrix A, das in der Zeile z und der Spalte s steht. Ist z eine ungerade Zahl und s eine gerade oder z gerade und s ungerade, wird das Teilergebnis mit -1 multipliziert. Für alle $s \in \{1,2,..,n\}$ summiert man diese Teilergebnisse auf und erhält die Determinante. Mathematisch ausgedrückt:

$$\det(A) := \sum_{s=1}^{n} (-1)^{s+z} * a_{z,s} * \det(A_{z.s})$$

Pierre Simon Marquis de Laplace (1749-1827)

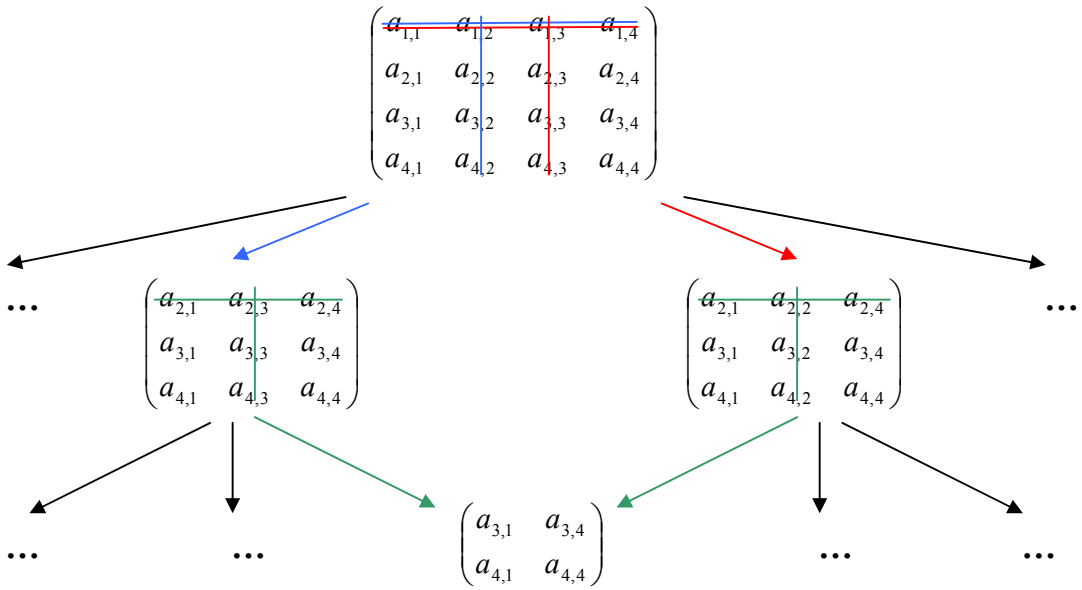
Carl Friedrich Gauß (1777-1855)

Es empfiehlt sich, z jeweils so zu wählen, dass möglichst viele $a_{z,s} = 0$ sind, weil man dann die Unterdeterminanten $\det(A_{z.s})$ nicht mehr berechnen muss (weil dann $a_{z,s} * \det(A_{z.s}) = 0 * \det(A_{z.s}) = 0$ ist).

Es existieren weitere Determinanten Gesetze, die es erlauben, die Matrix mit Hilfe des *Gauß’schen Eliminationsverfahrens* auf Diagonalenform zu bringen (wodurch man die Determinante dann praktisch ablesen kann), was mit einem Aufwand proportional zu n^3 iterativ funktioniert. Mit diesem Algorithmus beschäftige ich mich hier aber nicht, sondern mit der Optimierung des Laplace’schen Entwicklungsschemas.

Beschreibung des Topdown Vorverarbeitenden Determinanten Schemas (TVDS)

Der erste Gedanke hinter dem TVDS ist, dass es überflüssige Arbeit ist, jede Unterdeterminante in einer Matrix zu berechnen, denn wenn eine Unterdeterminante mehrmals vorkommt,



ist es sinnvoller, ihren Wert **abzuspeichern und bei Bedarf wieder abzurufen**, schließlich können auch größere Unterdeterminanten mehrfach vorkommen. Z.B. in einer 12x12 Matrix kann eine 10x10 Unterdeterminante zweimal vorkommen. Eine erneute Berechnung dieser Unterdeterminante würde 2.606.501 Arbeitsschritte bedeuten (vgl. S. 7), es sei denn man hat das Ergebnis der ersten Berechnung abgespeichert.

Wie speichert man nun eine Unterdeterminante ab, um sie bei Bedarf wieder zu finden? In der Matrix ist jede (komplette) Zeile und jede (komplette) Spalte entweder gestrichen oder nicht gestrichen, es gibt für jede Zeile und jede Spalte zwei mögliche Zustände, es bietet sich an, einen Boole'schen Wert für jede Zeile und Spalte anzulegen:

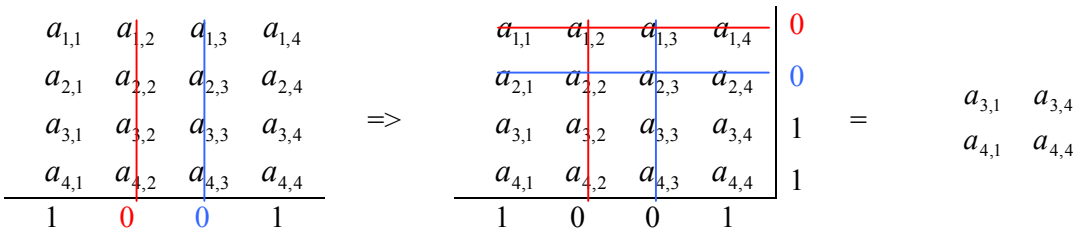
$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$	1
$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	$a_{2,4}$	0
$a_{3,1}$	$a_{3,2}$	$a_{3,3}$	$a_{3,4}$	1
$a_{4,1}$	$a_{4,2}$	$a_{4,3}$	$a_{4,4}$	1
1	1	0	1	

$=$ $\begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,4} \\ a_{3,1} & a_{3,2} & a_{3,4} \\ a_{4,1} & a_{4,2} & a_{4,4} \end{pmatrix}$

Man kann also die Unterdeterminante einer nxn Matrix (nachdem man sie einmal berechnet hat) festhalten indem man ihren Zahlenwert abspeichert, verknüpft mit einem 2n-Tupel von Boole'schen Werten (zur Identifizierung), im obigen Beispiel etwa 11011011.

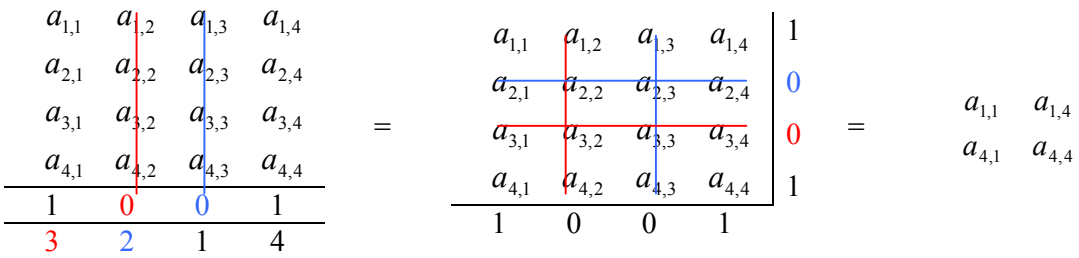
Der zweite Gedanke hinter dem TVDS ist die Überlegung, warum man ein 2n-Tupel benötigen sollte. Schränkt man das Laplace'sche Entwicklungsschema dahingehend ein, dass man nicht nach einer beliebigen Zeile entwickelt, sondern immer nach der obersten, ist die Unterdeterminante durch ein **n**-Tupel von Boole'schen Werten eindeutig definiert, denn die

Anzahl der (von oben) gestrichenen Zeilen entspricht dann der Anzahl der gestrichenen Spalten:

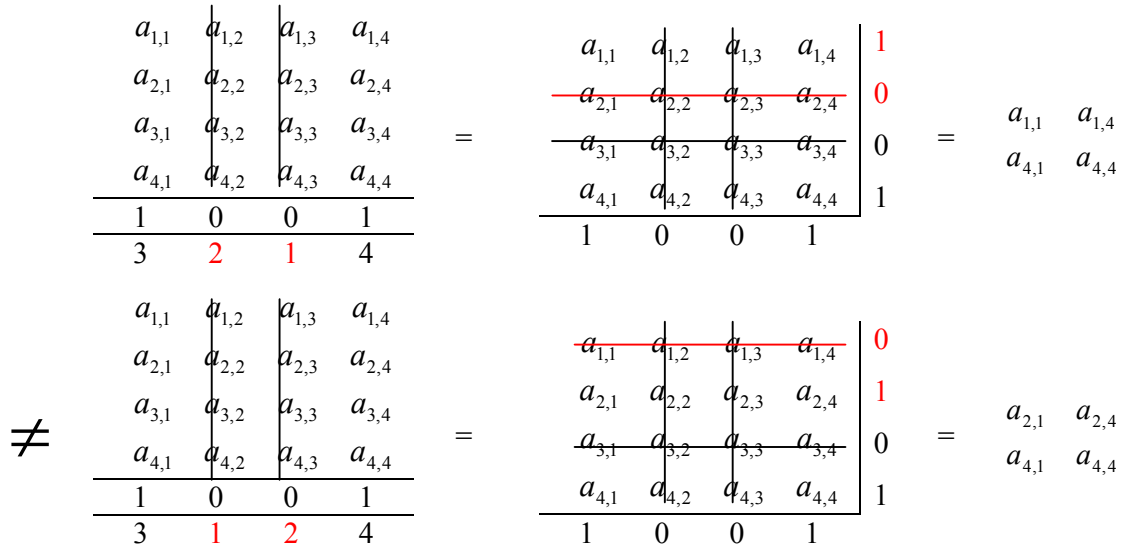


Wir können also eine hinreichende Anzahl von Unterdeterminanten durch ein **n**-Tupel von Boole'schen Zahlen repräsentieren.

Der dritte Gedanke hinter dem TVDS ist die Frage, warum man immer genau nach der obersten Zeile entwickeln sollte. Es genügt, eine festgelegte Reihenfolge zu haben. Die gestrichenen Zeilen ergeben sich dann aus der Anzahl der gestrichenen Spalten und der Entwicklungs-Reihenfolge. Ist z.B. eine Spalte gestrichen und ist die erste Zahl der Reihenfolge eine 3, bedeutet das, dass die dritte Zeile gestrichen ist:



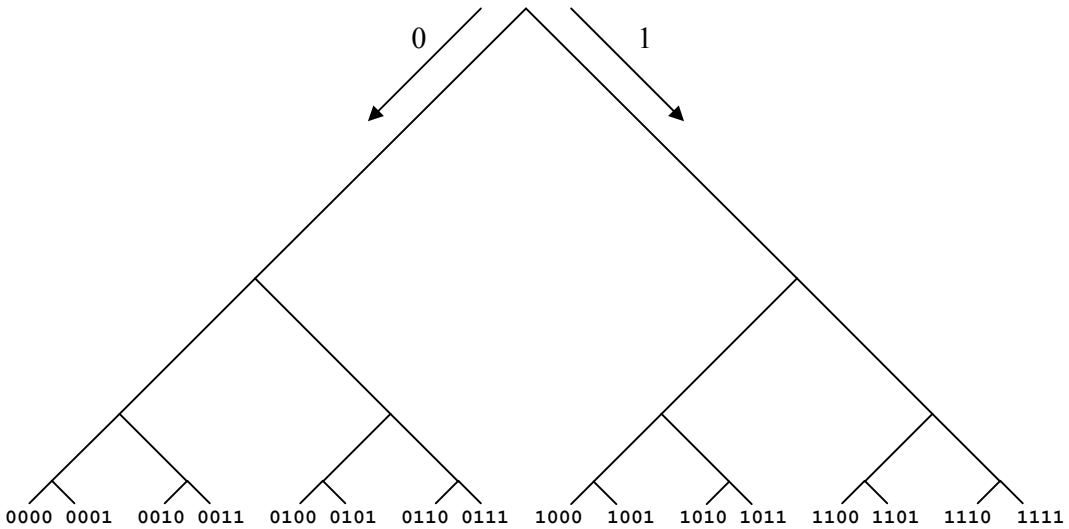
Die Reihenfolge darf (im Gegensatz zum Laplace Schema) während der Entwicklung nicht verändert werden, denn dann wäre die Darstellung der Untermatrix durch ein Boole'sches **n**-Tupel nicht mehr eindeutig.



Den Vorteil des Laplace Schemas, dass man sich immer die günstigste Zeile zum Entwickeln aussuchen kann (die mit den meisten Nullen), kann man sich also zumindest teilweise zunutze machen, indem man eine sinnvolle Reihenfolge aufstellt, nach der man entwickelt. Hier fängt man mit der Zeile, die die meisten Nullen enthält, an, fährt fort mit der Zeile, die die zweit meisten Nullen enthält etc. ahnlich wie man das Laplace Schema umsetzen würde, nur, dass man diese Reihenfolge nicht mehr in einzelnen Rekursionsstufen verändert, was beim Laplace Schema dann möglich ist, wenn (durch eine gestrichene Spalte) Nullen in späteren Zeilen wegfallen.

Der TVDS-Baum

Es stellt sich die Frage, wie man möglichst effektiv die berechneten Unterdeterminanten abspeichert, damit man sie mit möglichst geringem Zeitaufwand abrufen kann.
Abspeicherung in einer linearen Liste wäre sehr uneffektiv, da man bis zu $2^n - n - 1$ (vgl. S. 6) Elemente durchsuchen müsste, wobei zu jedem Element ein Boole'sches **n**-Tupel mit gespeichert werden müsste, das immer mit dem gesuchten Tupel verglichen werden müsste. Im schlimmsten Fall bedeutete dies einen Suchaufwand von n * $(2^n - n - 1)$, was absolut inakzeptabel ist.
Man könnte die Binärzahl in eine Dezimalzahl umrechnen und anhand dieser einen Wert aus einem Array lesen (Hashing), allerdings wird der Index eines Arrays in vielen Programmiersprachen durch eine Integer Zahl verarbeitet, man kann also nicht mehr als 2^{16} Elemente in einem Array verwalten (weil eine Integer Zahl 2 Byte = 16 Bit groß ist) was zur Folge hätte, dass man maximal die Determinante einer 16x16 Matrix berechnen könnte. Anm.: Borland C++ 5 ist aus Gründen der Benutzerfreundlichkeit, aber entgegen ANSI Norm standardmäßig so eingestellt, dass eine Integer Zahl 4 Byte = 32 Bit groß ist, hier könnte man die Determinante einer 32x32 Matrix berechnen, dieser code wäre aber nicht portabel!
Aufgrund der binären Struktur (dem Boole'schen n-Tupel) durch die die Untermatrizen identifiziert werden, bietet sich ein Binärer Suchbaum an. Dieser wird nun folgendermaßen aufgebaut: von dem aktuell gewählten Knoten auf der Ebene i geht man zum rechten Nachfolger, falls das i-te Element des Tupels eine 1 ist, ansonsten zum linken Nachfolger. In n Schritten hat man dann eine n stellige Binärzahl aufgebaut.



So entspricht jedes Blatt des Baumes genau einer eindeutig definierten Binärzahl. Man kann also die berechneten Unterdeterminanten an den Blättern des Baumes abspeichern und mit einem Aufwand proportional zu n (weil der Baum genau die Tiefe n hat) abrufen.

Leistungsanalyse anhand einer Aufwandsabschätzung

(Bestimmung von Formeln zur Berechnung der nötigen Rekursionsaufrufe)

- Die Anzahl der möglichen Boole'schen n-Tupel ist gleich 2^n, es müssen also maximal 2^n verschiedene Unterdeterminanten berechnet werden, aber n-Tupel, die nur eine 1 beinhalten werden nicht berücksichtigt, weil man schon bei einer 2x2 Matrix die Determinante explizit berechnet und ihre 1x1 Unterdeterminanten nicht mehr zu „berechnen" sind. Es wird daher die Anzahl der n-Tupel mit nur einer 1 (n Stück) vom Aufwand abgezogen: $2^n - n$. Das n-Tupel, das nur aus 0en besteht wird ebenfalls nicht berechnet: $2^n - n - 1$. Diese Formel ist zu offensichtlich, als dass man sie noch beweisen würde.
- Ohne Speicherung der Ergebnisse sind zu berechnen:
  - eine nxn Matrix $\left(1 = \frac{n!}{n!}\right)$
  - für jede der n gestrichenen Spalten eine (n-1)x(n-1) Untermatrix $\left(n = \frac{n!}{(n-1)!}\right)$
  - Für jede der (n-1) gestrichenen Spalten der n Untermatrizen eine (n-2)x(n-2) Unter-Untermatrix $\left(n*(n-1) = \frac{n!}{(n-2)!}\right)$
  - Für jede der (n-2) gestrichenen Spalten der n*(n-1) Unter-Untermatrizen eine (n-3)x(n-3) Unter-Unter-Untermatrix $\left(n*(n-1)*(n-2) = \frac{n!}{(n-3)!}\right)$ usw.

  Es fällt auf, dass die Anzahl der zu berechnenden ixi Matrizen immer $\frac{n!}{i!}$ beträgt, die Anzahl **aller** Determinanten, die zur Berechnung der Determinante einer nxn Matrix nötig sind ergibt sich nun aus der Summe der Anzahlen von nötigen ixi Matrizen für alle $i \in \{2,3,...,n\}$ (1x1 Matrizen gehen nicht in die Rechnung ein, weil 2x2 Matrizen explizit berechnet werden und damit ihre 1x1 „Unterdeterminanten" überflüssig sind) Die Formel lautet also $\sum_{i=2}^{n} \frac{n!}{i!}$

  Nachdem nun die Formel argumentativ plausibel gemacht wurde, fehlt noch ein mathematischer Beweis für ihre Korrektheit. Dieser wird per vollständiger Induktion erbracht:

  Induktionsanfang:
  Da R(2)=1 keinen aussagekräftigen Induktionsanfang darstellt, zeige ich R(3)

$$R(3) = \sum_{i=2}^{3} \frac{3!}{i!} = \frac{6}{2} + \frac{6}{6} = 4$$

  Es gibt 4 Rekursionsaufrufe, einen für die 3x3 Matrix und jeweils einen für die 3 Unterdeterminanten, die dann explizit berechnet werden.

  Induktionsschluss:
  Bei einer (n+1)x(n+1) Matrix gibt es (n+1) nxn Untermatrizen, die mit jeweils R(n) Rekursionsaufrufen berechnet werden müssen. Die Anzahl der Rekursionsaufrufe

ver-(n+1)-facht sich. Ausserdem wird ein weiterer Rekursionsaufruf nötig, für die (n+1)x(n+1) Matrix selbst. Induktionsschließen wir:

$$R(n+1) = (n+1) * R(n) + 1 = (n+1) * \left(\sum_{i=2}^{n} \frac{n!}{i!} \right) + 1$$

$$= \left(\sum_{i=2}^{n} \frac{(n+1)!}{i!} \right) + 1$$

$$1 = \frac{(n+1)!}{(n+1)!}$$

$$= \left(\sum_{i=2}^{n} \frac{(n+1)!}{i!} \right) + \frac{(n+1)!}{(n+1)!}$$

$$= \sum_{i=2}^{n+1} \frac{(n+1)!}{i!}$$

qed

Wir sehen, dass diese Formel korrekt ist, sie hat aber den Nachteil, dass es eine iterative Formel ist, besser wäre eine explizite:

$$\sum_{i=2}^{n} \frac{n!}{i!} = n! * \sum_{i=2}^{n} \frac{1}{i!}$$

$$\lim_{n->\infty} \sum_{i=0}^{n} \frac{1}{i!} = e \qquad \text{(Taylor Polynom)}$$

$$\lim_{n->\infty} \sum_{i=2}^{n} \frac{1}{i!} = \lim_{n->\infty} \sum_{i=0}^{n} \frac{1}{i!} - \sum_{i=0}^{1} \frac{1}{i!}$$

$$= e - 2$$

$$= n! * (e-2) \quad \textbf{(diese Formel ist bereits für n = 2 hinreichend genau)}$$

Vergleichen wir nun die nötigen Rekursionsaufrufe des TVDS und des Laplace Schemas:

n	Laplace $n!*(e-2)$	TVDS $2^n - n - 1$	TVDS : Laplace	Laplace : TVDS
3	4	4	100,00 %	100,00 %
4	17	11	64,71 %	154,55 %
5	86	26	30,23 %	330,77 %
6	517	57	11,03 %	907,02 %
7	3.620	120	3,31 %	3.016,67 %
8	28.961	247	0,85 %	11.725,10 %
9	260.650	502	0,19 %	51.922,31 %
10	2.606.501	1.013	0,04 %	257.305,13 %
11	28.671.512	2.036	0,007 %	1.408.227,50 %

Beispielaufgabe

Da eine Demonstration anhand einer 5x5 Matrix allein 4 Seiten umfassen würde, wird das TVDS hier anhand einer 4x4 Matrix demonstriert, auch, wenn man seine Leistungsfähigkeit daran nicht sehr gut erkennen kann, weil hier die TVDS-Cuts nur bei 2x2 Matrizen vorgenommen werden, die ansonsten explizit berechnet würden, aber was warum passiert kann man hieran hoffentlich erkennen. Eine Demonstration der Leistungsfähigkeit, bei der visuell dargestellt wird, schnell die Anzahl der wegfallenden Berechnungen durch die TVDS-Cuts steigt, findet sich auf Seite 9.

```
4 6 5 7
8 5 8 6
4 3 8 4
7 8 7 9
--------
1 1 1 1 ist unbekannt

      5 8 6
      3 8 4
      8 7 9
      --------
      0 1 1 1 ist unbekannt

            8 4
            7 9
            --------
            0 0 1 1 ist unbekannt
            2x2 Determinante: 44

            3 4
            8 9
            --------
            0 1 0 1 ist unbekannt
            2x2 Determinante: -5

            3 8
            8 7
            --------
            0 1 1 0 ist unbekannt
            2x2 Determinante: -43

      0 1 1 1 fertig berechnet (2)

      8 8 6
      4 8 4
      7 7 9
      --------
      1 0 1 1 ist unbekannt

            8 4
            7 9
            --------
            0 0 1 1 ist bekannt (44)

            4 4
            7 9
            --------
            1 0 0 1 ist unbekannt
            2x2 Determinante: 8

            4 8
            7 7
            --------
            1 0 1 0 ist unbekannt
            2x2 Determinante: -28

1 0 1 1 fertig berechnet (120)

8 5 6
4 3 4
7 8 9
--------
1 1 0 1 ist unbekannt

      3 4
      8 9
      --------
      0 1 0 1 ist bekannt (-5)

      4 4
      7 9
      --------
      1 0 0 1 ist bekannt (8)

      4 3
      7 8
      --------
      1 1 0 0 ist unbekannt
      2x2 Determinante: 11

1 1 0 1 fertig berechnet (-14)

8 5 8
4 3 8
7 8 7
--------
1 1 1 0 ist unbekannt

      3 8
      8 7
      --------
      0 1 1 0 ist bekannt (-43)

      4 8
      7 7
      --------
      1 0 1 0 ist bekannt (-28)

      4 3
      7 8
      --------
      1 1 0 0 ist bekannt (11)

      1 1 1 0 fertig berechnet (-116)

1 1 1 1 fertig berechnet (30)
```

Ersparnisdemonstration

Im Gegensatz zur Beispielaufgabe zeige ich hier nur die binären Darstellungen. Hierdurch erkennt man, wie schnell die Anzahl der übersprungenen Berechnungen ansteigt, dass jedes dieser Booleschen n-Tupel tatsächlich bereits vorgekommen ist und jedes n-Tupel tatsächlich Nachfolger seines Vorgängers ist. Man beachte: die Anzahl der Boole'schen 6-Tupel ist 517, die Anzahl der rot markierten Boole'schen 6-Tupel ist 57 (vgl. S. 7)

```
111111
  011111
    001111
      000111
        000011
        000101
        000110
      001011
        000011
        001001
        001010
      001101
        000101
        001001
        001100
      001110
        000110
        001010
        001100
    010111
      000111
        000011
        000101
        000110
      010011
        000011
        010001
        010010
      010101
        000101
        010001
        010100
      010110
        000110
        010010
        010100
    011011
      001011
        000011
        001001
        001010
      010011
        000011
        010001
        010010
      011001
        001001
        010001
        011000
      011010
        001010
        010010
        011000
    011101
      001101
        000101
        001001
        001100
      010101
        000101
        010001
        010100
      011001
        001001
        010001
        011000
      011100
        001100
        010100
        011000
    011110
      001110
        000110
        001010

        001100
      010110
        000110
        010010
        010100
      011010
        001010
        010010
        011000
      011100
        001100
        010100
        011000
  101111
    001111
      000111
        000011
        000101
        000110
      001011
        000011
        001001
        001010
      001101
        000101
        001001
        001100
      001110
        000110
        001010
        001100
    100111
      000111
        000011
        000101
        000110
      100011
        000011
        100001
        100010
      100101
        000101
        100001
        100100
      100110
        000110
        100010
        100100
    101011
      001011
        000011
        001001
        001010
      100011
        000011
        100001
        100010
      101001
        001001
        100001
        101000
      101010
        001010
        100010
        101000
    101101
      001101
        000101
        001001
        001100
      100101
        000101
        100001
        100100

      101001
        001001
        100001
        101000
      101100
        001100
        100100
        101000
    101110
      001110
        000110
        001010
        001100
      100110
        000110
        100010
        100100
      101010
        001010
        100010
        101000
      101100
        001100
        100100
        101000
  110111
    010111
      000111
        000011
        000101
        000110
      010011
        000011
        010001
        010010
      010101
        000101
        010001
        010100
      010110
        000110
        010010
        010100
    100111
      000111
        000011
        000101
        000110
      100011
        000011
        100001
        100010
      100101
        000101
        100001
        100100
      100110
        000110
        100010
        100100
    110011
      010011
        000011
        010001
        010010
      100011
        000011
        100001
        100010
      110001
        010001
        100001
        110000
      110010

        010010
        100010
        110000
    110101
      010101
        000101
        010001
        010100
      100101
        000101
        100001
        100100
      110001
        010001
        100001
        110000
      110100
        010100
        100100
        110000
    110110
      010110
        000110
        010010
        010100
      100110
        000110
        100010
        100100
      110010
        010010
        100010
        110000
      110100
        010100
        100100
        110000
  111011
    011011
      001011
        000011
        001001
        001010
      010011
        000011
        010001
        010010
      011001
        001001
        010001
        011000
      011010
        001010
        010010
        011000
    101011
      001011
        000011
        001001
        001010
      100011
        000011
        100001
        100010
      101001
        001001
        100001
        101000
      101010
        001010
        100010
        101000
    110011
      010011

        000011
        010001
        010010
      100011
        000011
        100001
        100010
      110001
        010001
        100001
        110000
      110010
        010010
        100010
        110000
    111001
      011001
        001001
        010001
        011000
      101001
        001001
        100001
        101000
      110001
        010001
        100001
        110000
      111000
        011000
        101000
        110000
    111010
      011010
        001010
        010010
        011000
      101010
        001010
        100010
        101000
      110010
        010010
        100010
        110000
      111000
        011000
        101000
        110000
  111101
    011101
      001101
        000101
        001001
        001100
      010101
        000101
        010001
        010100
      011001
        001001
        010001
        011000
      011100
        001100
        010100
        011000
    101101
      001101
        000101
        001001
        001100
      100101
        000101

        100001
        100100
      101001
        001001
        100001
        101000
      101100
        001100
        100100
        101000
    110101
      010101
        000101
        010001
        010100
      100101
        000101
        100001
        100100
      110001
        010001
        100001
        110000
      110100
        010100
        100100
        110000
    111001
      011001
        001001
        010001
        011000
      101001
        001001
        100001
        101000
      110001
        010001
        100001
        110000
      111000
        011000
        101000
        110000
    111100
      011100
        001100
        010100
        011000
      101100
        001100
        100100
        101000
      110100
        010100
        100100
        110000
      111000
        011000
        101000
        110000
  111110
    011110
      001110
        000110
        001010
        001100
      010110
        000110
        010010
        010100
      011010
        001010
        010010

        011000
      011100
        001100
        010100
        011000
    101110
      001110
        000110
        001010
        001100
      100110
        000110
        100010
        100100
      101010
        001010
        100010
        101000
      101100
        001100
        100100
        101000
    110110
      010110
        000110
        010010
        010100
      100110
        000110
        100010
        100100
      110010
        010010
        100010
        110000
      110100
        010100
        100100
        110000
    111010
      011010
        001010
        010010
        011000
      101010
        001010
        100010
        101000
      110010
        010010
        100010
        110000
      111000
        011000
        101000
        110000
    111100
      011100
        001100
        010100
        011000
      101100
        001100
        100100
        101000
      110100
        010100
        100100
        110000
      111000
        011000
        101000
        110000
```

unbekannt	bekannt (=> gespart)	gespart

Quellcode (in c++)

```
int dimension;
int** matrix;
bool* n_tupel;
int* reihenfolge;
TVDS_tree* calculated = NULL;
int Bestimme_Reihenfolge()                                   Reihenfolge bestimmen
   {
   int* anzahl;
   for(int i = 0; i < dimension; i++)
      {
      anzahl[i] = 0;
      reihenfolge[i] = i;
      for(int j = 0; j < dimension; j++)                     Nullen zählen
         if(matrix[i][j] == 0)
            anzahl[i]++;
      }
   int faktor = 1;                                                    (1)
   for(int i = 1; i < dimension; i++)                  Anzahlen sortieren (bubblesort)
      for(int j = 0; j < dimension - i; j++)
         if(anzahl[j] < anzahl[j+1])
            {
            swap(anzahl[j], anzahl[j+1]);
            swap(reihenfolge[j], reihenfolge[j+1]);
            faktor *= -1;
            }
   return(faktor);
   }
int TVDS(int rekdepth = 0)
   {
   int ret = 0;
   int* leafcont = findleaf(calculated, n_tupel);                  TVDS-Cut
   if(leafcont != NULL)
      return(*(leafcont));
   TVDS_tree* leaf = lastleaf;                  siehe “implementierung des TVDS-Baumes“
   if(rekdepth == dimension-2)                  2x2 Determinante explizit berechnen
      {
      int sa; for(sa = 0; sa < dimension; sa++) if(n_tupel[sa]) break;
      int sb; for(sb = sa+1; sb < dimension; sb++) if(n_tupel[sb]) break;
      int za = reihenfolge[dimension-2];
      int zb = reihenfolge[dimension-1];
      ret = matrix[za][sa]*matrix[zb][sb]-matrix[za][sb]*matrix[zb][sa];
      leaf->content = new(int);                    berechnete Determinante speichern
      *(leaf->content) = ret;
      return(ret);
      }
   int z = reihenfolge[rekdepth];               Zeile aus der Reihenfolge auslesen
   int faktor = 1;                                                    (2)
   for(int s = 0; s < dimension; s++)                  Schleife über die Spalten
      if(n_tupel[s])                            wenn Spalte nicht gestrichen ist
         {
         if(matrix[z][s] != 0)
            {
            n_tupel[s] = false;                                Spalte streichen
            ret += faktor * matrix[z][s] * TVDS(rekdepth+1);          Rekursion
            n_tupel[s] = true;                                 Spalte freigeben
            }
         faktor *= -1;
         }
   leaf->content = new(int);                  fertig bestimmte Determinante speichern
   *(leaf->content) = ret;
   return(ret);                                 (Unter-)Determinante zurückgeben
   }
```

Implementierung des TVDS-Baumes

```
class TVDS_tree                                                    Klassen Deklaration
   {
   public:
      TVDS_tree* lnext;                                  Zeiger auf linken Nachfolger
      TVDS_tree* rnext;                                  Zeiger auf rechten Nachfolger
      int* content;                                      Inhalt (wird nur in Blättern verwendet)
      TVDS_tree()                                                          Konstruktor
         {
         lnext = NULL;
         rnext = NULL;
         content = NULL;
         }
      ~TVDS_tree()                                                         Destruktor
         {
         if(rnext != NULL)
            delete(rnext);
         if(lnext != NULL)
            delete(lnext);
         }
   };

TVDS_tree* lastleaf;
int* findleaf(TVDS_tree* run, bool* ident)                                        (3)
   {
   for(int i=0; i<dimension; i++)                     Schleife über die Elemente des Tupels
      {                                               zur iterativen Traversierung des Baumes
      if(*ident)
         {
         if(run->rnext == NULL)                      Bei Bedarf den Baum weiter aufbauen (4)
            run->rnext = new(TVDS_tree);
         run = run->rnext;                                  rechten Nachfolger besuchen
         }
      else
         {
         if(run->lnext == NULL)                      Bei Bedarf den Baum weiter aufbauen
            run->lnext = new(TVDS_tree);
         run = run->lnext;                                  linken Nachfolger besuchen
         }
      ident++;
      }
   lastleaf = run;
   return(run->content);                  Zeiger auf das Blatt (Knoten auf Ebene n) zurückgeben
   }
```

(1) & (2) Diese Faktoren sind dazu gut, die Berechnung $(-1)^{z+s}$ zu umgehen, sie beruhen auf der Tatsache, dass auf jede gerade Zahl eine ungerade folgt und auf jede ungerade eine gerade, deshalb ist $(-1)^{s+z} = -\left((-1)^{(s-1)+z}\right)$ und deshalb wird bei (2), um $(-1)^{s+z}$ zu erhalten, einfach $(-1)^{(s-1)+z}$ negiert. Um das jeweilis erste $(-1)^{s+z}$ (in Abhängigkeit von z, das je nach Entwicklungsreihenfolge gerade oder ungerade sein kann) nicht berechnen zu müssen, nutzt man in (1) eins der auf S.2 erwähnten Determinanten Gesetze, nämlich, dass wenn man zwei Zeilen in einer Matrix vertauscht, sich ihre Determinante ver (-1)facht. Man berechnet die Determinante so, als lägen ihre Zeilen in Entwicklungsreihenfolge untereinander (weshalb dann das erste z = 1 ist, und weil $(-1)^{s+z}$ auch $= -\left((-1)^{s+(z-1)}\right)$ ist, braucht kein $(-1)^{s+z}$ mehr berechnet zu werden), dann muss die Determinante nur noch mit (-1) multipliziert werden, falls eine ungerade Anzahl von Zeilen vertauscht wurde - und genau das wird bei (1) geprüft.

(3) Weil man einen Zeiger auf eine Integer Zahl als Inhalt des Blattes verwendet, kann man ohne eine zusätzliche Bool'sche Variable überprüfen, ob der Inhalt des Blattes bereits existiert oder nicht (dadurch, dass man prüft, ob der Zeiger noch auf „NULL", also ins „Nichts" zeigt). In der findleaf Funktion wird dieser Zeiger zurückgegeben und ein Zeiger auf das Blatt, in dem er sich befindet in „lastleaf" gespeichert, so kann man den Inhalt des Blattes überprüfen, ohne über den Zeiger auf das Blatt den Inhalt dereferenzieren zu müssen. Sollte der Inhalt noch nicht existieren, kann man ihn (nachdem man ihn berechnet hat) über den Zeiger auf das Blatt abspeichern, ohne das Blatt in einer insert Prozedur erneut zu suchen. Diese Methode, den Zeiger auf den Inhalt zurückzugeben und den Zeiger auf das Blatt an anderer Stelle zu speichern hat sich empirisch als die Effektivste erwiesen.

(4) Dadurch, dass man den Baum während der Suche nach den Blättern aufbaut, braucht man ihn nicht vorher mit einem Aufwand von 2^n aufzubauen, was einen zusätzlichen Zeitaufwand von ca. 100% der Laufzeit des TVDS mit sich bringen würde.

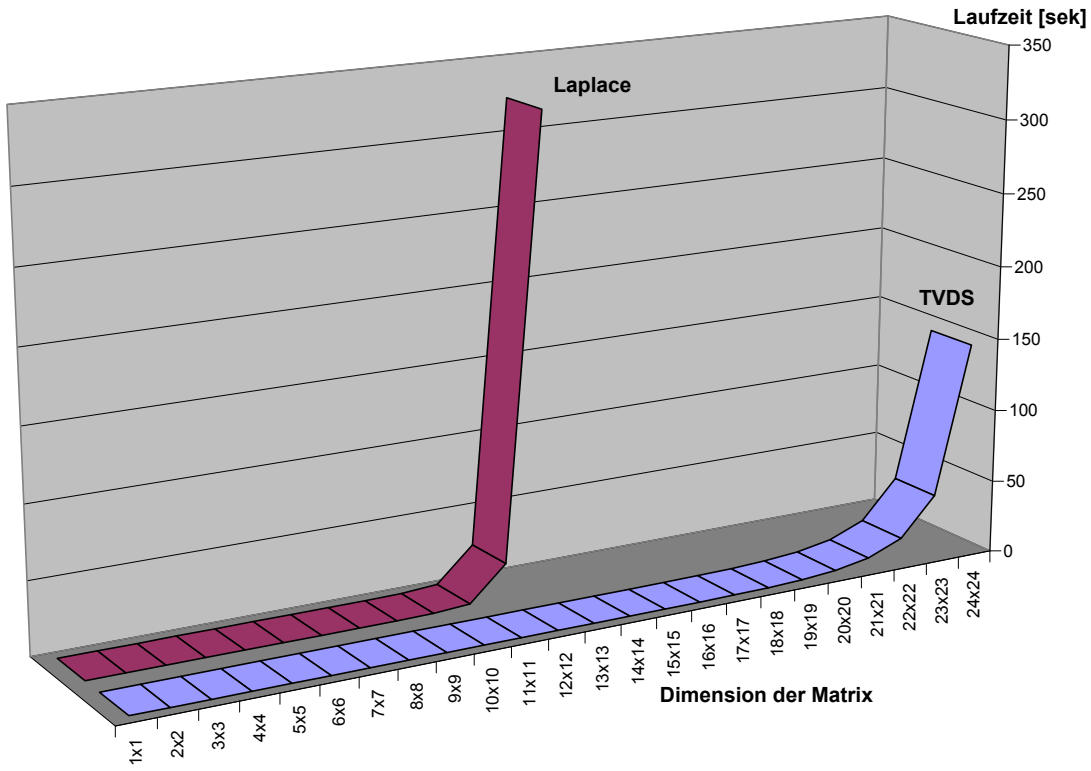
Testläufe

```
Korrektheitstest                      16.10.2002    20:27
-----------------------------------------------------------
Determinanten Berechnet nach TVDS und Laplace:  1000000
Übereinstimmende Ergebnisse:                    1000000
Unterschiedliche Ergebnisse:                          0
```

```
Aufwands-Korrektheits Test              08.11.2002    17:04
-----------------------------------------------------------
  n    |  n!*(e-2)   |   Rlp(n)    | (2^n)-n-1   |  Rtvds(n)
-------+-------------+-------------+-------------+-----------
 3x3   |           4 |           4 |           4 |           4
 4x4   |          17 |          17 |          11 |          11
 5x5   |          86 |          86 |          26 |          26
 6x6   |         517 |         517 |          57 |          57
 7x7   |        3620 |        3620 |         120 |         120
 8x8   |       28961 |       28961 |         247 |         247
 9x9   |      260650 |      260650 |         502 |         502
10x10  |     2606501 |     2606501 |        1013 |        1013
11x11  |    28671512 |    28671512 |        2036 |        2036
12x12  |   344058145 |   344058145 |        4083 |        4083
13x13  |  4472755886 |  4472755886 |        8178 |        8178
14x14  |             |             |       16369 |       16369
15x15  |             |             |       32752 |       32752
16x16  |             |             |       65519 |       65519
17x17  |             |             |      131054 |      131054
18x18  |             |             |      262125 |      262125
19x19  |             |             |      524268 |      524268
20x20  |             |             |     1048555 |     1048555
21x21  |             |             |     2097130 |     2097130
22x22  |             |             |     4194281 |     4194281
23x23  |             |             |     8388584 |     8388584
24x24  |             |             |    16777191 |    16777191
```

Leistungsanalyse anhand von Zeitmessungen

Auf einem P4 2,4GHz



Die Laufzeiten der Algorithmen sind proportional zu ihren Rekursionsaufrufen, dadurch lässt sich folgende Gleichung zur rechnerischen Bestimmung der Laufzeit des Laplace Schemas für eine 25x25 Matrix aufstellen:

$$\frac{t_{LP}(25)}{t_{LP}(13)} = \frac{R_{LP}(25)}{R_{LP}(13)}$$

Setzt man nun die Messdaten und die errechneten Rekursionsaufrufe ein, erhält man

$$\frac{t_{LP}(25)}{324sek} = \frac{11.141.420.304.415.739.821.629.898}{4.472.755.886}$$

$$\Leftrightarrow t_{LP}(25) \approx 324sek * 2.490.952.018.930.670$$

$$\Leftrightarrow t_{LP}(25) \approx 807.068.454.133.537.235 \text{ sek}$$

$$\Leftrightarrow t_{LP}(25) \approx 25{,}6 \textit{ Milliarden} \text{ Jahre}$$

der Messwert der Laufzeit des TVDS für eine 25x25 Matrix beträgt 13:28 min!

Aussicht

Der nächste logische Schritt zur Weiterentwicklung des Algorithmus wäre, ihn nicht mehr Top-Down, sondern Bottom-Up arbeiten zu lassen, indem man aus dem TVDS Baum alle Boole'schen n-Tupel, die 2 Einsen enthalten auswählt, ihre Vorgänger bestimmt und zu diesen die Determinante die zu dem n-Tupel gehört addiert bzw. subtrahiert. Danach werden alle Vorgänger der n-Tupel mit 3 Einsen bestimmt usw.
Alle Überlegungen zur Verbesserung dieses Algorithmus sind aber eher Gedankenspiele, da er nie die Leistungsfähigkeit des Gauß'schen Eliminationsverfahrens erreichen wird. Seine Geschwindigkeit wird immer zu 2^n proportional bleiben und damit zwangsläufig (zumindest für größere n) langsamer sein als das Gauß Verfahren, das eine Laufzeit proportional zu n^3 hat und iterativ arbeitet.

Danksagungen

Ich danke

- Tatjana van de Sand dafür, dass sie immer für mich da ist
- Herrn Helmut Schumacher, dessen hervorragender Unterricht mich zur Entwicklung dieses Algorithmus befähigt hat
- Andreas Tiedge, mit dessen Hilfe ich 1997 meine ersten Programmierversuche gemacht habe.

Bild- und Literaturnachweis

Bilder:

Carl Friedrich Gauß: http://www.th.physik.uni-frankfurt.de/~jr/physlist.html
Pierre Simon Laplace: http://www.cielosur.com/archisamar.htm

Texte:

Die Determinante:
Springer Verlag - Jänich - Lineare Algebra, 8.Auflage, S. 135

Das Laplace'sche Entwicklungsschema ($\det(A) := \sum_{s=1}^{n} (-1)^{s+z} * a_{z,s} * \det(A_{z.s})$)
Springer Verlag - Jänich - Lineare Algebra, 8.Auflage, S. 138

$\lim_{n->\infty} \sum_{i=0}^{n} \frac{1}{i!} = e$
Klett Verlag - Sieber - Mathematische Formeln, Erweiterte Ausgabe E, S. 26