Institut für Systemarchitektur
Kanalkodierung, Leistungsnachweis 03.03.2008

Leistungsnachweis, WS 07/08

zur Aufgabenstellung

Faltungscodes

Programmtechnische Realisierung des Viterbi-Algorithmus

Bearbeiter: **Rist, Ronald**
Matr.-Nr. 3209815
St.-g. Informatik, Jg. 2005
e-Mail: s0200738@mail.inf.tu-dresden.de

Aufgabenstellung:

1. Dekodieren Sie die Empfangsfolge eines Faltungscodes

b = (111110011010111011001010111010010011000000011
1001111011011000000011111010111000011111010000
0001000100000000100001111000010)

mittels programmtechnischer Realisierung des VITERBI-Algorithmus und ermitteln Sie den Quelltext der Informationsfolge b*.
Der Faltungskodierer sei durch folgende Koeffizientenmatrix beschrieben:

$$G = \begin{bmatrix} 100111 \\ 101011 \\ 111101 \end{bmatrix}$$

Bei der Übertragung handelt es sich um alphanumerischen Text:
A=(00001), B=(00010), C=(00011), ..., ␣=(11011)

Vergleichen Sie Ihr Ergebnis mit d_f des Faltungscodes (siehe Vorlesung, OFD-Faltungscodes). Welche Schlussfolgerungen lassen sich daraus ableiten?

Lösung:

Die Quellcodefolge lautet

b* = (100101010111001011100111001110111100)

Dies entspricht der Buchstabenfolge

RUYNNN\

Wobei die geschätzte Kodefolge

b_{korr} = (111001011010111011100010010110101000110000000 01
100001101101100000001111101011100001111101011
111101010000000010000111100000 0)

ist, welche einen Abstand von 18 zur Empfangsfolge hat.

Die freie Distanz des Codes ist $d_f = 13$ (gemäß OFD-Tabelle, Eintrag für R = ⅓ (m = 3) und k=5, die dort angegebene Matrix G = $(47_8\ 53_8\ 75_8)^T$ entspricht der Matrix der Aufgabenstellung). Dies entspricht der Näherung $d_f = w(G) = 13$. Die Anzahl korrigierbarer Fehler liegt bei mindestens $f_k = \left\lfloor \frac{d_f - 1}{2} \right\rfloor = 6$ je Kodesequenz, je nach Struktur bis zu $d_f = 13$ hin.

Es werden aber vermutlich mehr Fehler bei der Übertragung aufgetreten sein, weswegen es sich vermutlich um eine inkorrekte Dekodierung handelt (die Originalfolge wird wohl einen noch größeren Abstand zur fehlerbehafteten Empfangsfolge haben)

Zum Anwendungsprogramm:

Die Anwendung ist mit Borland C++ 4.0 (für Windows) compiliert, sowie die Anwendungsoberfläche damit erstellt.

Die Kernfunktionalität ist in den Dateien VITERBI.cpp (Viterbi-Dekodierung und gewöhnliche Enkodierung) und ALPH.cpp (Wandlung 5bit zu Buchstaben) gekapselt. Zumindest diese sollten in anderen Compilern wiederverwendbar sein.
Ein Abzug dieser beiden Dateien findet sich im Anhang dieses Dokuments.

Aufgabenstellung:

2. Ist ein periodisches Auftreten von Fehlern im Abstand 5 K rekonstruierbar, die Anzahl der Fehler dabei von d_f abhängend?

Lösung:

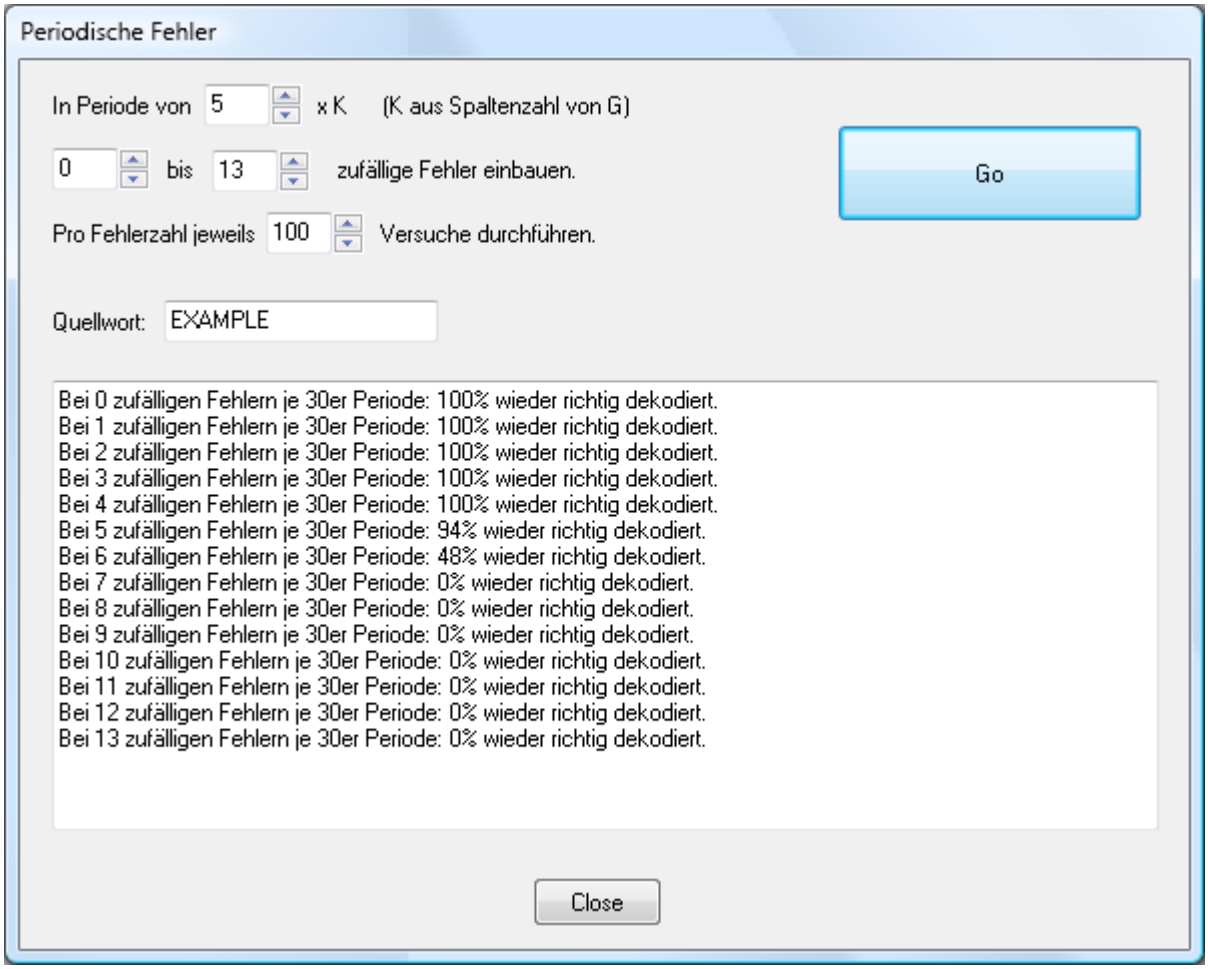
Das ist nur bedingt möglich (z.B. da ja auch zwischen den 5K-Blöcken keine Terminierung erfolgt) und stark vom jeweiligen Fehlermuster abhängig

Die sei zwei kleinen Beispielen demonstriert (5K = 5*6 = 30):
Text: EXAMPLE (7 Buchstaben)
a* = (00101110000000101101100000110000101)
a = (00000011100110001101100000010 0.00111100000011100110001110011 0
1010110010110011111111110010110.011001000001100100101010110111 1)
↯
b = (00100111110010010000110000000100.00111110000000101010011110011 0
1011100010111010101111100111 10.011001000101001111010111011 1)
b* = (00101110000000101101100000110000101)
Text: EXAMPLE
(f_k = 6 Bitfehler je 30 Bit Block, 1 und 2 Stellen lang, Folge korrekt dekodiert)

Text: EXAMPLE (7 Buchstaben)
a* = (00101110000000101101100000110000101)
a = (00000011100110001101100000010 0.00111100000011100110001110011 0
1010110010110011111111110010110.011001000001100100101010110111 1)
↯
b = (00111011100111111011000000100.00110001100110100110001110011 0
1010110010110010001000100101100.110010000011010101100111110111 1)
b* = (00101110000000101101100001101001101)
Text: EXAMPZM
(f_k = 6 Bitfehler je 30 Bit Block, 1 bis 3 Stellen lang, Folge nicht korrekt dekodiert, wobei allein die Fehler in den hinteren beiden Blöcken zur Fehlkorrektur führen)

Programmtechnische Umsetzung:

Es gibt also auch Fälle, in denen 6 Bitfehler je 30 Bit Block nicht mehr zu korrigieren sind. Im Programm ist über den Button [periodische Fehler ...] (dieser erscheint, wenn "Fehler einbauen" aktiviert wird) ein Unterfenster erreichbar:

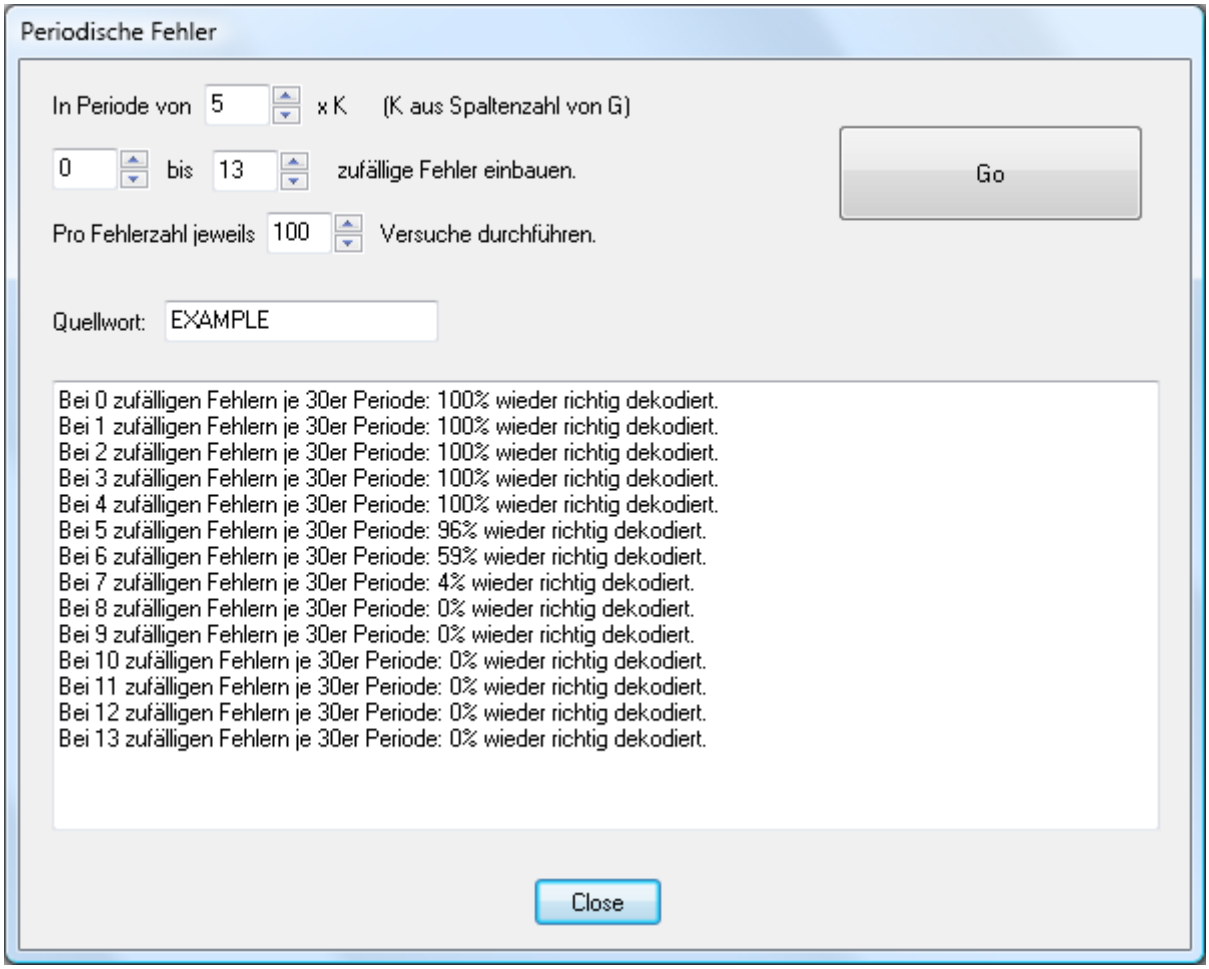


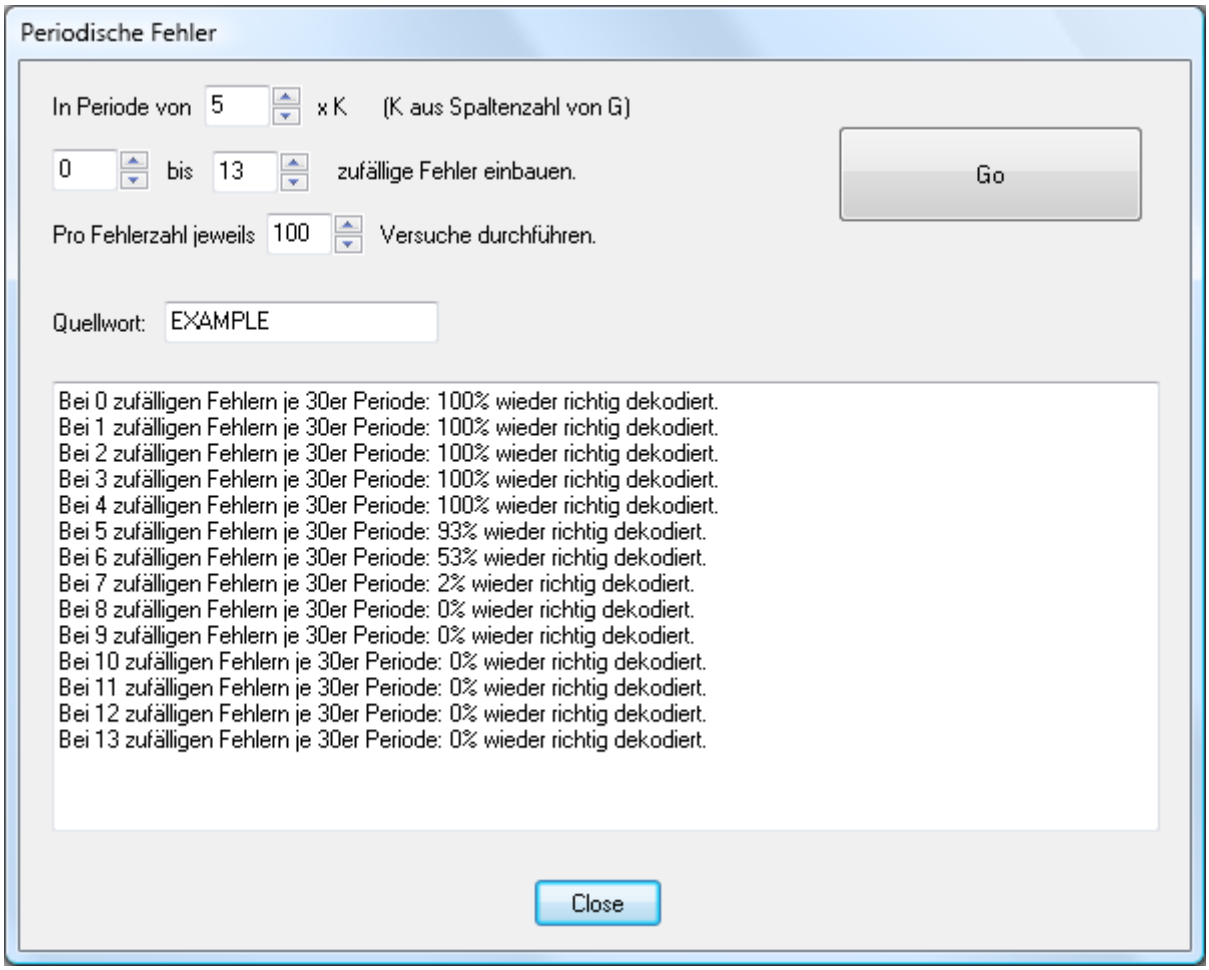
Damit ist es möglich, zufällig generierte Fehlermuster in den gewünschten Längen im Rahmen der Einstellungen im oberen Teil des Fensters auf die Folge (hier das Wort "EXAMPLE") zu legen. Das Ergebnis ist eine kleine statistische Erhebung, die deutlich macht, dass zwar noch fast alle Folgen mit 5 Fehlern je 30 Bit Block, im Schnitt nur die Hälfte der Kodefolgen mit 6 Fehlern je Block wieder richtig dekodiert werden können.

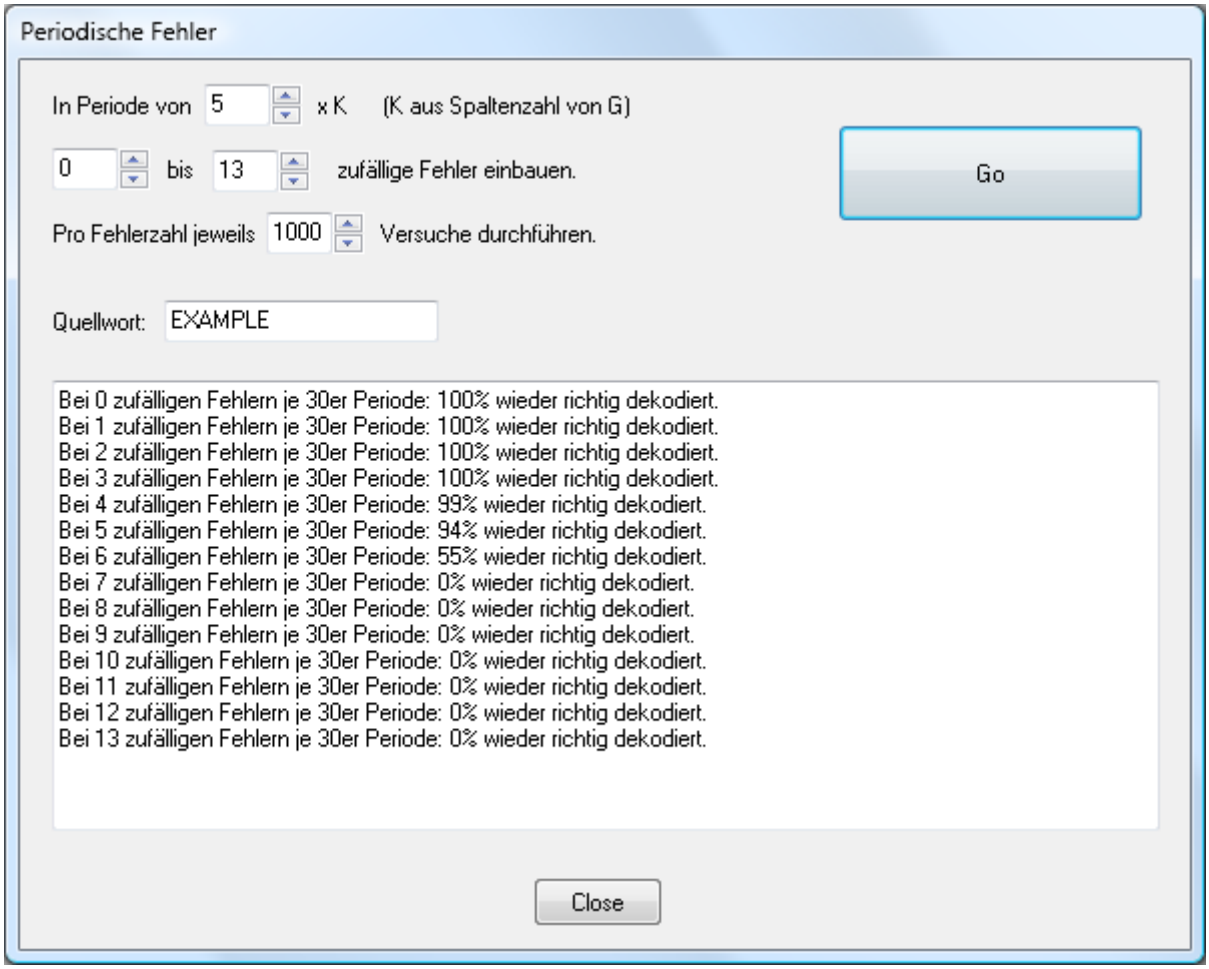
Interessant ist auch, dass scheinbar mitunter sogar ein paar wenige Folgen mit 7 Fehlern je Block rekonstruiert werden können (das entspricht insgesamt immerhin 28 Fehlern in der ganzen Folge).

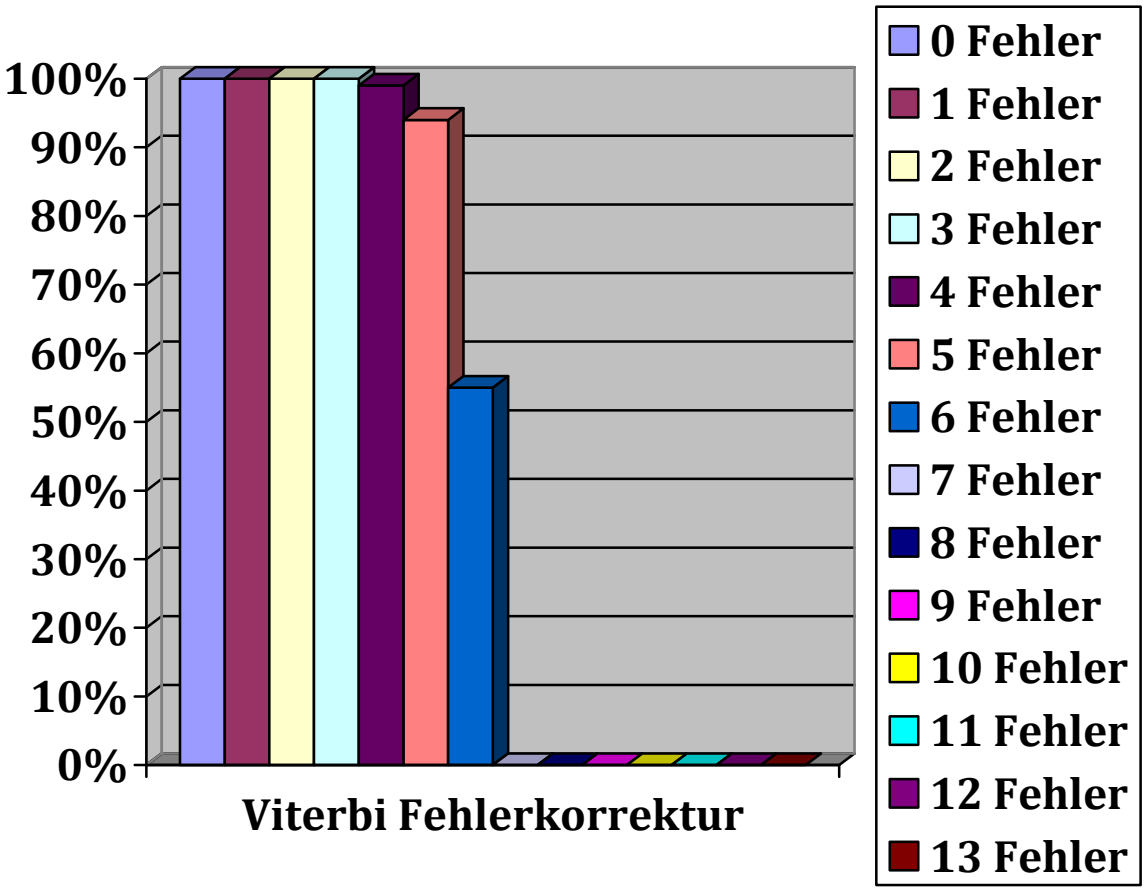
Diese traten jedoch nur in wenigen Durchläufen auf und gehen in einer größeren Messreihe (1000 Durchläufe pro Fehlerzahl, siehe unten) wieder unter.

(Hinweis: Es werden die Generatormatrix G und ggf. Punktierungsmatrix P verwendet, die im Hauptfenster eingegeben sind, d.h. die Ergebnisse beziehen sich auf genau die eingegebene Matrizen-Kombination. Hier im Beispiel ist G gemäß Aufgabenstellung und kein P genutzt.)


Periodische Fehler
In Periode von 5 x K (K aus Spaltenzahl von G)
0 bis 13 zufällige Fehler einbauen.
Go
Pro Fehlerzahl jeweils 100 Versuche durchführen.
Quellwort: EXAMPLE
Bei 0 zufälligen Fehlern je 30er Periode: 100% wieder richtig dekodiert.
Bei 1 zufälligen Fehlern je 30er Periode: 100% wieder richtig dekodiert.
Bei 2 zufälligen Fehlern je 30er Periode: 100% wieder richtig dekodiert.
Bei 3 zufälligen Fehlern je 30er Periode: 100% wieder richtig dekodiert.
Bei 4 zufälligen Fehlern je 30er Periode: 100% wieder richtig dekodiert.
Bei 5 zufälligen Fehlern je 30er Periode: 96% wieder richtig dekodiert.
Bei 6 zufälligen Fehlern je 30er Periode: 59% wieder richtig dekodiert.
Bei 7 zufälligen Fehlern je 30er Periode: 4% wieder richtig dekodiert.
Bei 8 zufälligen Fehlern je 30er Periode: 0% wieder richtig dekodiert.
Bei 9 zufälligen Fehlern je 30er Periode: 0% wieder richtig dekodiert.
Bei 10 zufälligen Fehlern je 30er Periode: 0% wieder richtig dekodiert.
Bei 11 zufälligen Fehlern je 30er Periode: 0% wieder richtig dekodiert.
Bei 12 zufälligen Fehlern je 30er Periode: 0% wieder richtig dekodiert.
Bei 13 zufälligen Fehlern je 30er Periode: 0% wieder richtig dekodiert.
Close


Periodische Fehler
In Periode von 5 x K (K aus Spaltenzahl von G)
0 bis 13 zufällige Fehler einbauen.
Go
Pro Fehlerzahl jeweils 100 Versuche durchführen.
Quellwort: EXAMPLE
Bei 0 zufälligen Fehlern je 30er Periode: 100% wieder richtig dekodiert.
Bei 1 zufälligen Fehlern je 30er Periode: 100% wieder richtig dekodiert.
Bei 2 zufälligen Fehlern je 30er Periode: 100% wieder richtig dekodiert.
Bei 3 zufälligen Fehlern je 30er Periode: 100% wieder richtig dekodiert.
Bei 4 zufälligen Fehlern je 30er Periode: 100% wieder richtig dekodiert.
Bei 5 zufälligen Fehlern je 30er Periode: 93% wieder richtig dekodiert.
Bei 6 zufälligen Fehlern je 30er Periode: 53% wieder richtig dekodiert.
Bei 7 zufälligen Fehlern je 30er Periode: 2% wieder richtig dekodiert.
Bei 8 zufälligen Fehlern je 30er Periode: 0% wieder richtig dekodiert.
Bei 9 zufälligen Fehlern je 30er Periode: 0% wieder richtig dekodiert.
Bei 10 zufälligen Fehlern je 30er Periode: 0% wieder richtig dekodiert.
Bei 11 zufälligen Fehlern je 30er Periode: 0% wieder richtig dekodiert.
Bei 12 zufälligen Fehlern je 30er Periode: 0% wieder richtig dekodiert.
Bei 13 zufälligen Fehlern je 30er Periode: 0% wieder richtig dekodiert.
Close


Periodische Fehler
In Periode von 5 x K (K aus Spaltenzahl von G)
0 bis 13 zufällige Fehler einbauen.
Go
Pro Fehlerzahl jeweils 1000 Versuche durchführen.
Quellwort: EXAMPLE
Bei 0 zufälligen Fehlern je 30er Periode: 100% wieder richtig dekodiert.
Bei 1 zufälligen Fehlern je 30er Periode: 100% wieder richtig dekodiert.
Bei 2 zufälligen Fehlern je 30er Periode: 100% wieder richtig dekodiert.
Bei 3 zufälligen Fehlern je 30er Periode: 100% wieder richtig dekodiert.
Bei 4 zufälligen Fehlern je 30er Periode: 99% wieder richtig dekodiert.
Bei 5 zufälligen Fehlern je 30er Periode: 94% wieder richtig dekodiert.
Bei 6 zufälligen Fehlern je 30er Periode: 55% wieder richtig dekodiert.
Bei 7 zufälligen Fehlern je 30er Periode: 0% wieder richtig dekodiert.
Bei 8 zufälligen Fehlern je 30er Periode: 0% wieder richtig dekodiert.
Bei 9 zufälligen Fehlern je 30er Periode: 0% wieder richtig dekodiert.
Bei 10 zufälligen Fehlern je 30er Periode: 0% wieder richtig dekodiert.
Bei 11 zufälligen Fehlern je 30er Periode: 0% wieder richtig dekodiert.
Bei 12 zufälligen Fehlern je 30er Periode: 0% wieder richtig dekodiert.
Bei 13 zufälligen Fehlern je 30er Periode: 0% wieder richtig dekodiert.
Close


100%
90%
80%
70%
60%
50%
40%
30%
20%
10%
0%
Viterbi Fehlerkorrektur
0 Fehler
1 Fehler
2 Fehler
3 Fehler
4 Fehler
5 Fehler
6 Fehler
7 Fehler
8 Fehler
9 Fehler
10 Fehler
11 Fehler
12 Fehler
13 Fehler

Aufgabenstellung:

3. Hat die Anzahl punktierter Stellen und deren Anordnung, z.B. für w(P) = const, Einfluss auf die Leistungsfähigkeit der Dekodierung?

Lösung:

Da Punktierung immer auf Kosten der Fehlerkorrekturfähigkeit geht, hat die Anzahl punktierter Stellen natürlich erheblichen Einfluss.
Da die Länge der Punktierungsmatrizen P nicht unbedingt mit der Einflusslänge des Mutterkodes übereinstimmen (veränderliche Lage der Stellen), müssen nun u.U. längere Kodefolgen als beim Mutterkode allein betrachtet werden, um die freie Distanz d_f zu bestimmen (bedingt durch die Punktierung können längere Kodesequenzen wieder geringere Gewichte haben als kürzere). Die Lösung sollte sich in k+2+p Übergängen finden lassen (wobei p die Anzahl der Spalten der Punktierungsmatrix sei). Der Aufwand wächst exponentiell mit der Anzahl der betrachteten Übergänge.
Die Anordnung der Punktierungsstellen ist ebenfalls entscheidend, da sich d_f je nach Lage der Punktierungsstellen erheblich unterscheiden kann (z.B. eine Spalte mit nur Nullen in P löscht ein Element ganz aus). Es kann außerdem zu einem katastrophalen Kodierer kommen.

Kleines Beispiel:

$$G = \begin{bmatrix} 101 \\ 111 \end{bmatrix}$$ (d.h. k = 2, m = 2, d_f = 5)

für $$P = \begin{bmatrix} 1001 \\ 1110 \end{bmatrix}$$ ist nur noch d_f = **2**, z.B.

a_1* = (00010100)	a_1 = (00 00 00 11 01 00 01 11)	a_{1P} = (00 00 10 10 11)
a_2* = (00111000)	a_2 = (00 00 11 10 01 10 11 00)	a_{2P} = (00 01 10 10 10)
a_3* = (00000000)	a_3 = (00 00 00 00 00 00 00 00)	a_{3P} = (00 00 00 00 00)
a_4* = (00101100)	a_4 = (00 00 11 01 00 10 10 11)	a_{4P} = (00 01 00 00 01)

(entspricht der Angabe im OFD)

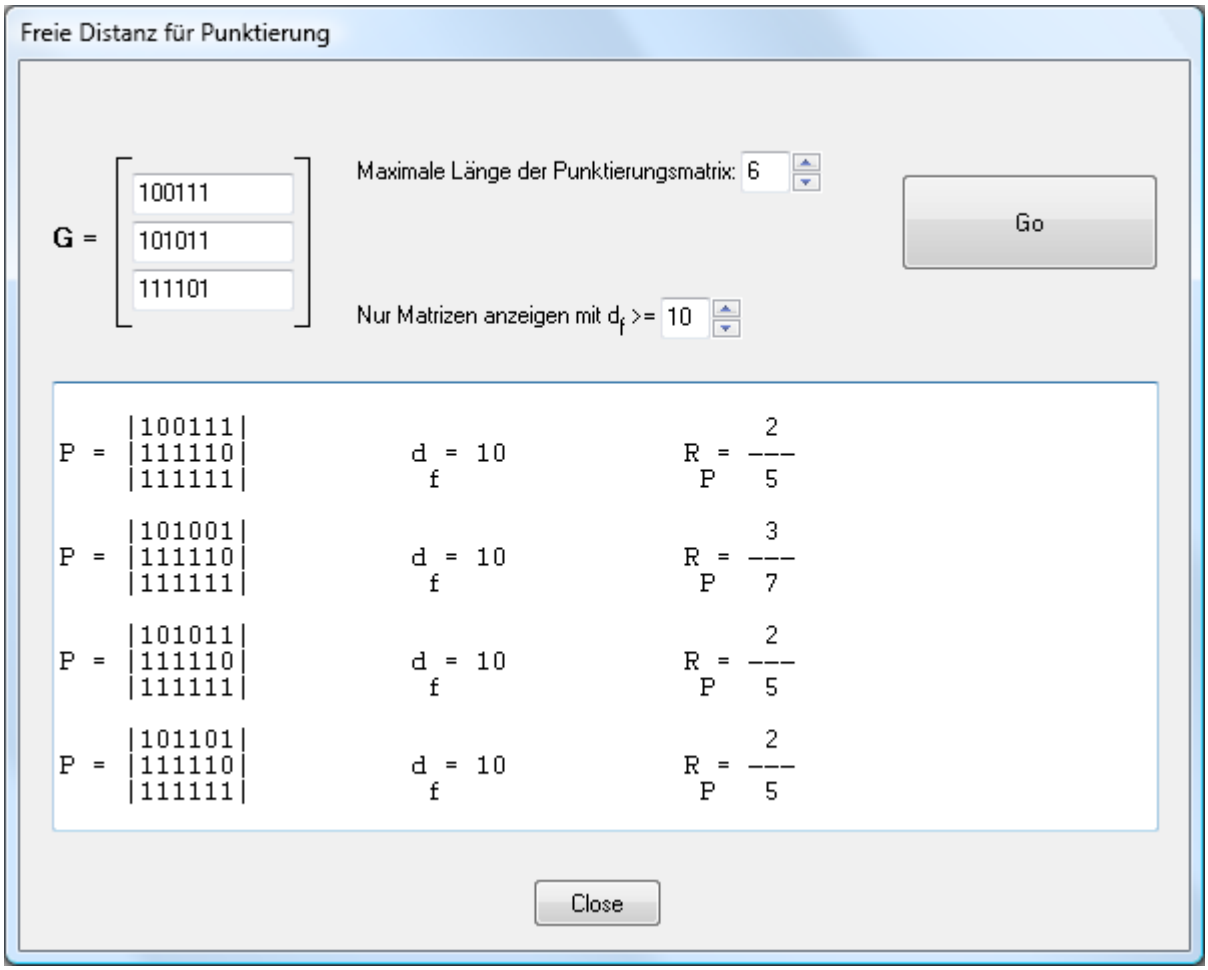
für $$P = \begin{bmatrix} 1110 \\ 1100 \end{bmatrix}$$ (gleiches Gewicht) ist sogar nur noch d_f = **1**, z.B.

a_1* = (00000000)	a_1 = (00 00 00 00 00 00 00 00)	a_{1P} = (00 00 00 00 00)
a_2* = (00010100)	a_2 = (00 00 00 11 01 00 01 11)	a_{2P} = (00 00 00 10 00)

Programmtechnische Umsetzung:

Die Funktionalität zur Bestimmung der verbleibenden freien Distanz d_f nach Punktierung ist in der Datei FDISTANZ.cpp gekapselt (siehe Anhang).

Über den Button [Punktierung/Analyse ...] (dieser erscheint, wenn "Punktierung" aktiviert wird) kann ein Unterfenster erreicht werden:



Damit können zu gegebener Generatormatrix Punktierungsmatrizen bis zu einer vorgegebenen bestimmten Länge (d.h. Anzahl Spalten) generiert und auf ihre freie Distanz d_f nach Anwendung auf den vorgegebenen Kodierer maschinell überprüft werden.
(Achtung: ab 5 Spalten dauert der Vorgang je nach Rechnergeschwindigkeit schon etwas länger.)

Die Ausgabe kann mit Angabe eines minimal zu erreichenden d_f gefiltert werden.

Das Programm baut das Trellis-Diagramm mit Übergangsbewertungen und damit ein Baumdiagramm (Speicherung der Pfade zum Auslesen und Punktieren anstatt ständiger Neuberechnung) auf und iteriert alle möglichen Wege zum Auffinden der Kodefolge mit nach der Punktierung minimalem Gewicht.

Folgende Ergebnisse waren festzustellen:

In den meisten Fällen gilt, dass Punktierungsmatrizen gleichen Gewichts (und gleicher Länge, d.h. also gleicher Koderate) auch die gleiche freie Distanz d_f hatten (hier nur ein Beispiel):

```
    |01|                              2
P = |11|          d = 10          R = ---
    |11|           f                P   5

    |10|                              2
P = |11|          d = 10          R = ---
    |11|           f                P   5                          Beispiel 1
```

Allerdings gibt es mitunter eine Abweichung von d_f um bis zu zwei (bei gleichem Gewicht und gleicher Länge):

```
    |101|                             1
P = |100|          d = 6           R = ---
    |111|           f                P   2

    |110|                             1
P = |100|          d = 7           R = ---
    |111|           f                P   2

    |001|                             1
P = |101|          d = 8           R = ---
    |111|           f                P   2                          Beispiel 2
```

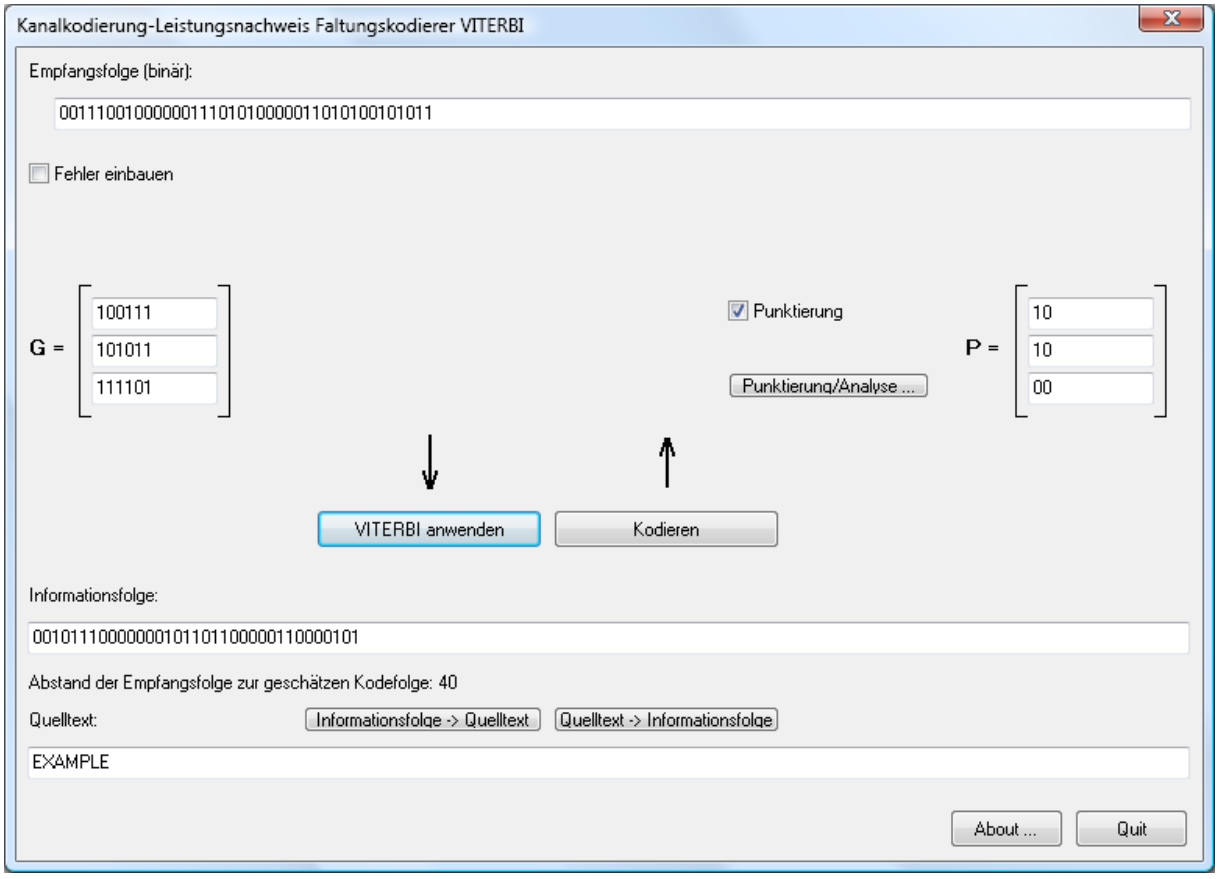
Daraus folgt im Umkehrschluss, dass manche Matrizen das gleiche d_f haben wie Matrizen höheren Gewichts (schlechterer Rate) [vgl. dazu auch Screenshot vorige Seite]:

```
    |0111|                            4
P = |1100|          d = 10          R = ---
    |1111|           f                P   9

    |1111|                            2
P = |1100|          d = 10          R = ---
    |1111|           f                P   5                          Beispiel 3

...

    |10100|                           5
P = |11111|          d = 10          R = ----
    |11111|           f                P   12

    |10101|                           5
P = |11111|          d = 10          R = ----
    |11111|           f                P   13                         Beispiel 4

...

    |100111|                          2
P = |111110|          d = 10          R = ---
    |111111|           f                P   5

    |101001|                          3
P = |111110|          d = 10          R = ---
    |111111|           f                P   7                          Beispiel 5
```

Abschließend ist bei stark verbesserter Koderate immer ein Verlust an freier Distanz festzustellen (vgl. Beispiel 2 der vorigen Seite).
Wenn d_f hingegen konstant gehalten werden soll (Beispiel 3 bis 5), verbessert sich die Koderate offensichtlich nur noch minimal bei länger werdenden Punktierungsmatrizen, womit diese also nicht unbedingt große Vorteile gegenüber kürzeren Matrizen haben.

Die Punktierung selbst kann mit dem Programm ebenfalls getestet werden (nachdem die CheckBox "Punktierung" aktiviert wurde), die Funktionalität dafür wird von der modifizierten Datei VITERBIMOD.cpp zur Verfügung gestellt (siehe Anhang).

Es kann dann z.B. eine vorher mit dem "Freie Distanz"-Unterfenster gefundene Punktierungsmatrix eingegeben und ihre Leistungsfähigkeit getestet werden.



Ablauf für dieses Beispiel:

1. Eingabe des Quelltextes "EXAMPLE", Klick auf [Quelltext -> Informationsfolge]
2. Eingabe der Punktierungsmatrix
3. Klick auf [Kodieren] liefert die punktierte (in dem Fall stark verkürzte) Empfangsfolge
4. Klick auf [VITERBI anwenden] liefert die Informationsfolge und das Quelltextwort, in diesem Fall trotz des hohen Abstandes zur geschätzten Kodefolge, korrekt zurück.

Jedoch kann (unter Nutzung dieser Punktierungsmatrix) bei nur einem zufällig auftretenden Fehler im Mittel nur noch bei 13% der Kodefolgen die Informationsfolge rekonstruiert werden (getestet mit der Funktion für periodische Fehler).

Anhang

VITERBI.cpp (Viterbi-Dekodierung und gewöhnliche Enkodierung, Kernfunktionalität)

```
#include <alloc.h>
#include <vcl.h>
#include "VITERBI.h"

/* Dekodierung mittels VITERBI-Algorithmus */
/* Leistungsnachweis, Ronald Rist          */

String Decode(String eingabe, String matrix1, String matrix2, String matrix3, char& Metrik)
{
          String ausgabe;
        String reverse;
        unsigned char y_triplet;
        unsigned char v[2][3];
        unsigned char k;
        int laenge;

        unsigned char *distanz;
        unsigned char *survivor;
        unsigned char *v_u_trellis_map;
        unsigned char h1, h2;

        //Matrixpuffer
        if ((strlen(matrix1.c_str())!=strlen(matrix2.c_str()))||(strlen(matrix2.c_str())!=strlen(matrix3.c_str()))) { ShowMessage("Bitte Matrix G auf
gleiche Zeilenlängen überprüfen."); return ""; }

        k = strlen(matrix1.c_str())-1; //z.B. G hat 6 Spalten -> k=5 innere Zustände
        laenge = strlen(eingabe.c_str())/3;
        distanz  = (unsigned char *) malloc((laenge+1)*(1<<k)); //Eingabelänge / 3 * 2^k
        survivor = (unsigned char *) malloc((laenge+1)*(1<<k)); //Eingabelänge / 3 * 2^k


        // Trellis:

        //Generierung einer "Map", um ein verkürztes Trellis zum "Nachschlagen"
        //(direkte Auswahl) zu haben, statt ständiger Neuberechnung
        //
        //Form: sigma(neuer Wert) | sigma'(alter Wert)_1 | sigma'(alter Wert)_2 | v_1 | v_2 | u_1 | u_2
        //      (entspr. Stelle)    (implizit)             (implizit)             m bit m bit 1 bit 1 bit -> seien 1 Char (m=3)
        //mit insgesamt 2^k Einträgen
        v_u_trellis_map = (unsigned char *) malloc(1<<k);

        for (unsigned int j=0; j<(1<<(k-1)); j++) //k = 5 -> 2^5 = 0..31, sigma(neu) iterieren
        {
           //Einzelkomponenten von v generieren:
           //hier erstmal 0..15 (führende 0 -> u=0),
           /*v ausnullen*/ for (int w1=0; w1<2; w1++) for (int w2=0; w2<3; w2++) v[w1][w2]=0;
           //p=0 (entspr. Matrix*u) fällt weg, da u=0
           for (unsigned int p=1; p<=k; p++)
           {
              //v1_1:
              v[0][0] = v[0][0] ^ (matrix1.c_str()[p]-48)*(((j<<1)&(1<<(k-p)))>>(k-p)); // ^ als XOR
                                                                                        // (j<<1) liefert sigma' (für 0 rausgeschoben)
                                                                                        // &(1<<(k-p)) liefert p-tes Bit von links
                                                                                        // >>p zurückshiften
              //v1_2:
              v[0][1] = v[0][1] ^ (matrix2.c_str()[p]-48)*(((j<<1)&(1<<(k-p)))>>(k-p));
              //v1_3:
              v[0][2] = v[0][2] ^ (matrix3.c_str()[p]-48)*(((j<<1)&(1<<(k-p)))>>(k-p));
              //v2_1:
              v[1][0] = v[1][0] ^ (matrix1.c_str()[p]-48)*((((j<<1)+1)&(1<<(k-p)))>>(k-p)); // (j<<1)+1 liefert sigma' (für 1 rausgeschoben)

              //v2_2:
              v[1][1] = v[1][1] ^ (matrix2.c_str()[p]-48)*((((j<<1)+1)&(1<<(k-p)))>>(k-p));
              //v2_3:
              v[1][2] = v[1][2] ^ (matrix3.c_str()[p]-48)*((((j<<1)+1)&(1<<(k-p)))>>(k-p));
           }

           //zusammensetzen:
           v_u_trellis_map[j]            = (v[0][0]<<7) + (v[0][1]<<6) + (v[0][2]<<5) + (v[1][0]<<4) + (v[1][1]<<3) + (v[1][2]<<2); // u_1 = u_2 = 0


           //Einzelkomponenten von v generieren:
           //nun für 16..31 (führende 1 -> u=1)
           /*v ausnullen*/ for (int w1=0; w1<2; w1++) for (int w2=0; w2<3; w2++) v[w1][w2]=0;
           //p=0 (entspr. Matrix*u) behandeln, da u=1
           for (int w1=0; w1<2; w1++) { v[w1][0] = matrix1.c_str()[0]-48; v[w1][1] = matrix2.c_str()[0]-48; v[w1][2] = matrix3.c_str()[0]-48; }
           for (unsigned char p=1; p<=k; p++)
           {
              //v1_1:
              v[0][0] = v[0][0] ^ (matrix1.c_str()[p]-48)*(((j<<1)&(1<<(k-p)))>>(k-p));
              //v1_2:
              v[0][1] = v[0][1] ^ (matrix2.c_str()[p]-48)*(((j<<1)&(1<<(k-p)))>>(k-p));
              //v1_3:
              v[0][2] = v[0][2] ^ (matrix3.c_str()[p]-48)*(((j<<1)&(1<<(k-p)))>>(k-p));

              //v2_1:
              v[1][0] = v[1][0] ^ (matrix1.c_str()[p]-48)*((((j<<1)+1)&(1<<(k-p)))>>(k-p));
              //v2_2:
              v[1][1] = v[1][1] ^ (matrix2.c_str()[p]-48)*((((j<<1)+1)&(1<<(k-p)))>>(k-p));
              //v2_3:
              v[1][2] = v[1][2] ^ (matrix3.c_str()[p]-48)*((((j<<1)+1)&(1<<(k-p)))>>(k-p));
           }
```

```
            //zusammensetzen:
            v_u_trellis_map[(1<<(k-1))+j] = (v[0][0]<<7) + (v[0][1]<<6) + (v[0][2]<<5) + (v[1][0]<<4) + (v[1][1]<<3) + (v[1][2]<<2) + 3; // u_1 = u_2 = 1
         }


         // Viterbi-Algorithmus:

         //t=0 (i=0) umsetzen:
         distanz[0]=0;
         for (int w1=1; w1<(1<<k); w1++) distanz[w1]=(1<<k);        for (int w1=0; w1<(1<<k); w1++) survivor[w1]=0;
            for (unsigned int i=1; i<=laenge; i++)
         {
            y_triplet = ((eingabe.c_str()[3*(i-1)]-48)<<2)+((eingabe.c_str()[3*(i-1)+1]-48)<<1)+(eingabe.c_str()[3*(i-1)+2]-48);
            for (unsigned int j=0; j<(1<<(k-1)); j++) //k = 5 -> 2^5 = 0..31
            //aus j (entpr. sigma) folgen beide Vorgänger und ihre Übergangsbewertungen eindeutig gemäß G
            {
               //hier erstmal 0..15 (führende 0 -> u=0),
               h1 = distanz[(1<<k)*(i-1)+(j<<1)]   + hammingdist( (v_u_trellis_map[j]    >>5), y_triplet); // >>5 liefert erste 3 Bit
               h2 = distanz[(1<<k)*(i-1)+(j<<1)+1] + hammingdist(((v_u_trellis_map[j]&28)>>2), y_triplet); // liefert zweite 3 Bit

               distanz[(1<<k)*i+j]             = min(h1, h2);
               survivor[(1<<k)*i+j]            = min_sl(h1, h2); //select, liefert 0 od. 1 je nachdem

               //nun 16..31 -> führende 1 -> u=1
               //(führende 1 abschneiden ("rausshiften"), da k nicht notwenigerweise identisch zu Bitzahl int
               h1 = distanz[(1<<k)*(i-1)+(j<<1)]   + hammingdist( (v_u_trellis_map[(1<<(k-1))+j]    >>5), y_triplet); // >>5 liefert erste 3 Bit
               h2 = distanz[(1<<k)*(i-1)+(j<<1)+1] + hammingdist(((v_u_trellis_map[(1<<(k-1))+j]&28)>>2), y_triplet); // liefert zweite 3 Bit

               distanz[(1<<k)*i+(1<<(k-1))+j]  = min(h1, h2);
               survivor[(1<<k)*i+(1<<(k-1))+j] = min_sl(h1, h2);
            }
         }

         //min. Abstand zur Empfangsfolge in Metrik distanz[(1<<k)*laenge+0] gespeichert -> ausgeben über in/out-value
         Metrik = distanz[(1<<k)*laenge+0];


         //Zurückverfolgung des Survivors:
         int s = 0; //Ausgangspunkt ist sigma = (0...0)
         /* //Alternative: Keine Terminierung annehmen und Distanzminimum als Startwert für Survivor-Rückverfolgung annehmen:
         int min = laenge*3;
         for (int p=0; p<(1<<k); p++) //Minimum aus letzter Stelle
         {
            ShowMessage(AnsiString((unsigned int)distanz[(1<<k)*laenge+p]));
            if (distanz[(1<<k)*laenge+p]<=min)
            {
               min = distanz[(1<<k)*laenge+p];
               s = p;
            }
         }      */

         for (int i=0; i<laenge; i++)
         {
            reverse = reverse + AnsiString((char)((s>>(k-1))+48)); //1. Bit von sigma impliziert Eingabe u
            s = (s<<1)%(1<<k) + survivor[(1<<k)*(laenge-i)+s]; //vorheriger Zustand, abh. von Survivor (+/- 1)
         }

         ausgabe="";
         for (int i=0; i<laenge-k; i++) ausgabe = ausgabe + reverse[laenge-i]; //-k -> Terminierung entfernen

         free(v_u_trellis_map);
         free(distanz);
         free(survivor);

            return ausgabe;
}

unsigned char hammingdist(unsigned char s, unsigned char y)
//(nur 3-bit)
{
   unsigned char abstand = s^y;
   return (abstand>>2)+(abstand>>1)%2+abstand%2;
}

unsigned char min(unsigned char h1, unsigned char h2)
{
  if (h1>h2) return h2;
  else return h1;
}

unsigned char min_sl(unsigned char h1, unsigned char h2)
{
  if (h1>h2) return 1;
  else return 0;
}

String Encode(String eingabe, String matrix1, String matrix2, String matrix3)
{
            String ausgabe;
         unsigned char v[2][3];
         unsigned char k;
         int laenge;
         unsigned char *v_u_trellis_map;

         //Matrixpuffer
         if ((strlen(matrix1.c_str())!=strlen(matrix2.c_str()))||(strlen(matrix2.c_str())!=strlen(matrix3.c_str()))) { ShowMessage("Bitte Matrix G auf
gleiche Zeilenlängen überprüfen."); return ""; }

         k = strlen(matrix1.c_str())-1; //z.B. G hat 6 Spalten -> k=5 innere Zustände

         //Eingabe-/Ausgabepuffer
         for (int i=0; i<k; i++) eingabe = eingabe + "0"; //Terminierung anhängen
```

```
        laenge = strlen(eingabe.c_str());


        // Trellis:

        //Generierung einer "Map", um ein verkürztes Trellis zum "Nachschlagen"
        //(direkte Auswahl) zu haben, statt ständiger Neuberechnung
        //
        //Form: sigma(neuer Wert) | sigma'(alter Wert)_1 | sigma'(alter Wert)_2 | v_1 | v_2 | u_1 | u_2
        //      (entspr. Stelle)    (implizit)            (implizit)             m bit m bit 1 bit 1 bit -> seien 1 Char (m=3)
        //mit insgesamt 2^k Einträgen
        v_u_trellis_map = (unsigned char *) malloc(1<<k);

        for (unsigned int j=0; j<(1<<(k-1)); j++) //k = 5 -> 2^5 = 0..31, sigma(neu) iterieren
        {
           //Einzelkomponenten von v generieren:
           //hier erstmal 0..15 (führende 0 -> u=0),
           /*v ausnullen*/ for (int w1=0; w1<2; w1++) for (int w2=0; w2<3; w2++) v[w1][w2]=0;
           //p=0 (entspr. Matrix*u) fällt weg, da u=0
           for (unsigned int p=1; p<=k; p++)
           {
              //v1_1:
              v[0][0] = v[0][0] ^ (matrix1.c_str()[p]-48)*(((j<<1)&(1<<(k-p)))>>(k-p)); // ^ als XOR
                                                                                        // (j<<1) liefert sigma' (für 0 rausgeschoben)
                                                                                        // &(1<<(k-p)) liefert p-tes Bit von links
                                                                                        // >>p zurückshiften
              //v1_2:
              v[0][1] = v[0][1] ^ (matrix2.c_str()[p]-48)*(((j<<1)&(1<<(k-p)))>>(k-p));
              //v1_3:
              v[0][2] = v[0][2] ^ (matrix3.c_str()[p]-48)*(((j<<1)&(1<<(k-p)))>>(k-p));
              //v2_1:
              v[1][0] = v[1][0] ^ (matrix1.c_str()[p]-48)*((((j<<1)+1)&(1<<(k-p)))>>(k-p)); // (j<<1)+1 liefert sigma' (für 1 rausgeschoben)

              //v2_2:
              v[1][1] = v[1][1] ^ (matrix2.c_str()[p]-48)*((((j<<1)+1)&(1<<(k-p)))>>(k-p));
              //v2_3:
              v[1][2] = v[1][2] ^ (matrix3.c_str()[p]-48)*((((j<<1)+1)&(1<<(k-p)))>>(k-p));
           }

           //zusammensetzen:
           v_u_trellis_map[j]            = (v[0][0]<<7) + (v[0][1]<<6) + (v[0][2]<<5) + (v[1][0]<<4) + (v[1][1]<<3) + (v[1][2]<<2); // u_1 = u_2 = 0


           //Einzelkomponenten von v generieren:
           //nun für 16..31 (führende 1 -> u=1)
           /*v ausnullen*/ for (int w1=0; w1<2; w1++) for (int w2=0; w2<3; w2++) v[w1][w2]=0;
           //p=0 (entspr. Matrix*u) behandeln, da u=1
           for (int w1=0; w1<2; w1++) { v[w1][0] = matrix1.c_str()[0]-48; v[w1][1] = matrix2.c_str()[0]-48; v[w1][2] = matrix3.c_str()[0]-48; }
           for (unsigned char p=1; p<=k; p++)
           {
              //v1_1:
              v[0][0] = v[0][0] ^ (matrix1.c_str()[p]-48)*(((j<<1)&(1<<(k-p)))>>(k-p));
              //v1_2:
              v[0][1] = v[0][1] ^ (matrix2.c_str()[p]-48)*(((j<<1)&(1<<(k-p)))>>(k-p));
              //v1_3:
              v[0][2] = v[0][2] ^ (matrix3.c_str()[p]-48)*(((j<<1)&(1<<(k-p)))>>(k-p));

              //v2_1:
              v[1][0] = v[1][0] ^ (matrix1.c_str()[p]-48)*((((j<<1)+1)&(1<<(k-p)))>>(k-p));
              //v2_2:
              v[1][1] = v[1][1] ^ (matrix2.c_str()[p]-48)*((((j<<1)+1)&(1<<(k-p)))>>(k-p));
              //v2_3:
              v[1][2] = v[1][2] ^ (matrix3.c_str()[p]-48)*((((j<<1)+1)&(1<<(k-p)))>>(k-p));
           }

           //zusammensetzen:
           v_u_trellis_map[(1<<(k-1))+j] = (v[0][0]<<7) + (v[0][1]<<6) + (v[0][2]<<5) + (v[1][0]<<4) + (v[1][1]<<3) + (v[1][2]<<2) + 3; // u_1 = u_2 = 1
        }


        //eigentliche Enkodierung (nun sehr einfach, reines Auswählen):
        ausgabe="";
        unsigned char s = 0; unsigned char s_alt = 0;
        unsigned char vv = 0;
        for (unsigned int i=0; i<laenge; i++)
        {
           s_alt=s;
           s=(s+((eingabe.c_str()[i]-48)<<k))>>1;  //u = eingabe[i] , -48 wg. ASCII
           if ((s_alt&1)==0) vv = ( (v_u_trellis_map[s]    )>>5)); // >>5 liefert erste 3 Bit
           else              vv = (((v_u_trellis_map[s]&28)>>2));

           ausgabe = ausgabe + AnsiString((char)(((vv&4)>>2)+48));  //ausgabe[i*3  ] = ((vv&4)>>2)+48;
           ausgabe = ausgabe + AnsiString((char)(((vv&2)>>1)+48));  //ausgabe[i*3+1] = ((vv&2)>>1)+48;
           ausgabe = ausgabe + AnsiString((char)(( vv&1    )+48));  //ausgabe[i*3+2] = ( vv&1    )+48;
        }

        free(v_u_trellis_map);

           return ausgabe;
}
```

ALPH.cpp (Wandlung 5 Bit zu Buchstaben)

```
#include <alloc.h>
#include <vcl.h>
#include "ALPH.h"

/* Umwandlung 5bit-Code in alphanumerischen Text */
/* Leistungsnachweis, Ronald Rist                */

String Convert(String eingabe)
{
          String ausgabe = "";
        char buchstabe;

          for (unsigned int i=0; i<strlen(eingabe.c_str()); i=i+5)
        {
           buchstabe = (((((((eingabe.c_str()[i]-48)<<1)+(eingabe.c_str()[i+1]-48))<<1)+(eingabe.c_str()[i+2]-48))<<1)+(eingabe.c_str()[i+3]-
48))<<1)+(eingabe.c_str()[i+4]-48)+64;
           //Zahlenkonvertierung basierend auf Standard-ASCII
           if (buchstabe==91) buchstabe=32; //Sonderfall Leerzeichen
           ausgabe = ausgabe + AnsiString((char) buchstabe);
        }
          return ausgabe;
}

String ConvertBack(String eingabe)
{
          String ausgabe = "";
        char code[5];

          for (unsigned int i=0; i<strlen(eingabe.c_str()); i++)
        {
           code[0] = ( (eingabe.c_str()[i]-64)>>4   )+48;
           code[1] = (((eingabe.c_str()[i]-64)>>3)&1)+48;
           code[2] = (((eingabe.c_str()[i]-64)>>2)&1)+48;
           code[3] = (((eingabe.c_str()[i]-64)>>1)&1)+48;
           code[4] = ( (eingabe.c_str()[i]-64)    &1)+48;
           //Zahlenkonvertierung basierend auf Standard-ASCII
           if (eingabe.c_str()[i]-64==32) { code[0]=1; code[1]=1; code[2]=0; code[3]=1; code[4]=1; } //Sonderfall Leerzeichen
           ausgabe = ausgabe + AnsiString((char*) code);
        }
          return ausgabe;
}
```

FDISTANZ.cpp (Bestimmung der verbleibenden freien Distanz d_f nach Punktierung)

```
#include <alloc.h>
#include <vcl.h>
#include "FDISTANZ.h"

/* Maschinelle Distanzbestimmung punktierter Faltungscodes */
/* Leistungsnachweis, Ronald Rist                          */

//Datenstruktur für Baum
typedef struct baumstruktur *knoten;
typedef struct baumstruktur { char sigma;
                              char v;
                              knoten u0, u1;
                            } baumknoten;

int fdistanz(String p1, String p2, String p3, String matrix1, String matrix2, String matrix3)
{
        // Trellis:
        unsigned char *v_u_trellis_map;
        unsigned char v[2][3];
        unsigned char k = strlen(matrix1.c_str())-1; //z.B. G hat 6 Spalten -> k=5 innere Zustände
  unsigned char P = strlen(p1.c_str());        //z.B. P hat 6 Spalten -> P=6

        //Generierung einer "Map", um ein verkürztes Trellis zum "Nachschlagen"
        //(direkte Auswahl) zu haben, statt ständiger Neuberechnung
        //
        //Form: sigma(neuer Wert) | sigma'(alter Wert)_1 | sigma'(alter Wert)_2 | v_1 | v_2 | u_1 | u_2
        //      (entspr. Stelle)    (implizit)             (implizit)             m bit m bit 1 bit 1 bit -> seien 1 Char (m=3)
        //mit insgesamt 2^k Einträgen
        v_u_trellis_map = (unsigned char *) malloc(1<<k);

        for (unsigned int j=0; j<(1<<(k-1)); j++) //k = 5 -> 2^5 = 0..31, sigma(neu) iterieren
        {
           //Einzelkomponenten von v generieren:
           //hier erstmal 0..15 (führende 0 -> u=0),
           /*v ausnullen*/ for (int w1=0; w1<2; w1++) for (int w2=0; w2<3; w2++) v[w1][w2]=0;
           //p=0 (entspr. Matrix*u) fällt weg, da u=0
           for (unsigned int p=1; p<=k; p++)
           {
              //v1_1:
              v[0][0] = v[0][0] ^ (matrix1.c_str()[p]-48)*(((j<<1)&(1<<(k-p)))>>(k-p)); // ^ als XOR
                                                                                        // (j<<1) liefert sigma' (für 0 rausgeschoben)
                                                                                        // &(1<<(k-p)) liefert p-tes Bit von links
                                                                                        // >>p zurückshiften
              //v1_2:
              v[0][1] = v[0][1] ^ (matrix2.c_str()[p]-48)*(((j<<1)&(1<<(k-p)))>>(k-p));
              //v1_3:
              v[0][2] = v[0][2] ^ (matrix3.c_str()[p]-48)*(((j<<1)&(1<<(k-p)))>>(k-p));
              //v2_1:
              v[1][0] = v[1][0] ^ (matrix1.c_str()[p]-48)*((((j<<1)+1)&(1<<(k-p)))>>(k-p)); // (j<<1)+1 liefert sigma' (für 1 rausgeschoben)

              //v2_2:
              v[1][1] = v[1][1] ^ (matrix2.c_str()[p]-48)*((((j<<1)+1)&(1<<(k-p)))>>(k-p));
              //v2_3:
              v[1][2] = v[1][2] ^ (matrix3.c_str()[p]-48)*((((j<<1)+1)&(1<<(k-p)))>>(k-p));
           }

           //zusammensetzen:
           v_u_trellis_map[j]           = (v[0][0]<<7) + (v[0][1]<<6) + (v[0][2]<<5) + (v[1][0]<<4) + (v[1][1]<<3) + (v[1][2]<<2); // u_1 = u_2 = 0


           //Einzelkomponenten von v generieren:
           //nun für 16..31 (führende 1 -> u=1)
           /*v ausnullen*/ for (int w1=0; w1<2; w1++) for (int w2=0; w2<3; w2++) v[w1][w2]=0;
           //p=0 (entspr. Matrix*u) behandeln, da u=1
           for (int w1=0; w1<2; w1++) { v[w1][0] = matrix1.c_str()[0]-48; v[w1][1] = matrix2.c_str()[0]-48; v[w1][2] = matrix3.c_str()[0]-48; }
           for (unsigned char p=1; p<=k; p++)
           {
              //v1_1:
              v[0][0] = v[0][0] ^ (matrix1.c_str()[p]-48)*(((j<<1)&(1<<(k-p)))>>(k-p));
              //v1_2:
              v[0][1] = v[0][1] ^ (matrix2.c_str()[p]-48)*(((j<<1)&(1<<(k-p)))>>(k-p));
              //v1_3:
              v[0][2] = v[0][2] ^ (matrix3.c_str()[p]-48)*(((j<<1)&(1<<(k-p)))>>(k-p));

              //v2_1:
              v[1][0] = v[1][0] ^ (matrix1.c_str()[p]-48)*((((j<<1)+1)&(1<<(k-p)))>>(k-p));
              //v2_2:
              v[1][1] = v[1][1] ^ (matrix2.c_str()[p]-48)*((((j<<1)+1)&(1<<(k-p)))>>(k-p));
              //v2_3:
              v[1][2] = v[1][2] ^ (matrix3.c_str()[p]-48)*((((j<<1)+1)&(1<<(k-p)))>>(k-p));
           }

           //zusammensetzen:
           v_u_trellis_map[(1<<(k-1))+j] = (v[0][0]<<7) + (v[0][1]<<6) + (v[0][2]<<5) + (v[1][0]<<4) + (v[1][1]<<3) + (v[1][2]<<2) + 3; // u_1 = u_2 = 1
        }

  String pfad;
  int dfmin = 1000; //Startwert (Maximum)
  int w = 0;
  knoten wurzel = NULL;

  //Baum aufstellen (statt ständiger Pfadneuberechnung)
  generiereBaum(&wurzel, 0/*sigma*/, 0/*v*/, v_u_trellis_map, k, P+k+2/*maximale Suchtiefe*/);
  //(unter Wurzel ganzer Baum zugreifbar)
```

```
  //(punktierte) Pfade vergleichen (die auf 0...0 enden und nicht identisch sind)
  for (int i=1; i<(1<<(P+2)); i++) //i=1: Nullpfad auslassen
  //alle p+k+2 Pfade durchgehen, letzte k Schritte müssen Nullschritte sein
  //-> p+2 variable Schritte verbleiben
  {
    knoten kn = wurzel;
    pfad = "";
    for (int j=1; j<=(P+2); j++)
    //Folge auslesen, dabei gleich "punktieren" (bestimmte Stellen löschen)
    //erstmal für die p+2 ersten Stellen
    {
      if (p1.c_str()[j%P] == '1') pfad = pfad + AnsiString((char)((((kn->v)&4)>>2)+48));
      if (p2.c_str()[j%P] == '1') pfad = pfad + AnsiString((char)((((kn->v)&2)>>1)+48));
      if (p3.c_str()[j%P] == '1') pfad = pfad + AnsiString((char)(( (kn->v)&1    )+48));

      //die j-te Stelle von i bestimmt nun, wohin "abgebogen" werden muss
      //(genau: die (k+P+1-j)-te Stelle von i)
      //if ((((i&(1<<(k+p+2-j)))>>(k+p+2-j)) ==)
      if (((i>>(P+2-j))&1) == 0) kn = (kn->u0);
      else kn = (kn->u1);
    }
    for (int j=(P+3); j<=(P+k+3); j++) //+3, da mit 000 eingestiegen wird und letzter Ausgang erforderlich
    //Folge auslesen, dabei gleich "punktieren" (bestimmte Stellen löschen)
    //nun die letzten ersten Stellen, die Nullschritte sein müssen
    //(Aufteilung spart massiv Rechenaufwand)
    {
      if (p1.c_str()[j%P] == '1') pfad = pfad + AnsiString((char)((((kn->v)&4)>>2)+48));
      if (p2.c_str()[j%P] == '1') pfad = pfad + AnsiString((char)((((kn->v)&2)>>1)+48));
      if (p3.c_str()[j%P] == '1') pfad = pfad + AnsiString((char)(( (kn->v)&1    )+48));
      kn = (kn->u0);
    }
    //Nullzustand nun am Ende mit Sicherheit erreicht
    //Pfadgewicht bestimmen
    w = 0;
    for (int o=0; o<strlen(pfad.c_str()); o++)
    {
      if (pfad.c_str()[o]==49) w++;
    }
    if (w<dfmin) dfmin=w;
  }

 freeBaum(&wurzel);
 free(v_u_trellis_map);
 return dfmin;
}



void generiereBaum(knoten *s, char sigma, char v, char *trellis, char k, char tiefe)
{
  knoten neu;
  neu = (knoten) malloc(sizeof(baumknoten)); // neues Element anlegen
  neu->sigma = sigma;
  neu->v = v; //v gibt immer das v(t) an, das beim Erreichen des aktuellen sigma ausgegeben wurde
  neu->u0 = NULL;
  neu->u1 = NULL;
  *s = neu;

  //linke Seite weiter generieren und rechte Seite weiter generieren (rechte nur, wenn tiefe>k,
  //da letzte k Schritte Nullschritte sein müssen -> später keine Auswertung der anderen Äste)
  //(if-Konstruktion so, das möglichst wenige if's je Durchlauf abgefragt werden -> schneller)
  if (tiefe>k)      //linke + rechte Seite
  {
    if ((sigma&1) == 0) //0 hinten raus schieben
    {
      generiereBaum(&((*s)->u0),  sigma>>1            ,  trellis[ sigma>>1            ]    >>5, trellis, k, tiefe-1);
      generiereBaum(&((*s)->u1), (sigma>>1)+(1<<(k-1)),  trellis[(sigma>>1)+(1<<(k-1))]    >>5, trellis, k, tiefe-1);
    }
    else               //1 hinten raus schieben
    {
      generiereBaum(&((*s)->u0),  sigma>>1            , (trellis[ sigma>>1            ]&28)>>2, trellis, k, tiefe-1);
      generiereBaum(&((*s)->u1), (sigma>>1)+(1<<(k-1)), (trellis[(sigma>>1)+(1<<(k-1))]&28)>>2, trellis, k, tiefe-1);
    }
  }
  else if (tiefe>0) //nur linke Seite (die aber immer)
  {
    if ((sigma&1) == 0) //0 hinten rausgeschoben
      generiereBaum(&((*s)->u0),  sigma>>1            ,  trellis[ sigma>>1            ]    >>5, trellis, k, tiefe-1);
    else               //1 hinten raus geschoben
      generiereBaum(&((*s)->u0),  sigma>>1            , (trellis[ sigma>>1            ]&28)>>2, trellis, k, tiefe-1);
  }
}

void freeBaum(knoten *s)
{
 if (*s!=NULL)
 {
   freeBaum(&((*s)->u0));
   freeBaum(&((*s)->u1));
   free(*s);
 }
}
```

[...]

VITERBIMOD.cpp (Viterbi-Dekodierung und Enkodierung für Punktierung)

```
#include <alloc.h>
#include <vcl.h>
#include "VITERBIMOD.h"

/* Dekodierung mittels VITERBI-Algorithmus */
/* Leistungsnachweis, Ronald Rist          */

String DecodeMOD(String eingabe, String matrix1, String matrix2, String matrix3, char& Metrik, String p1, String p2, String p3)
{
          String ausgabe;
        String eingabe2 = "";;
        String reverse;
        unsigned char y_triplet;
        unsigned char v[2][3];
        unsigned char k;
        int laenge;
        int laenge2;

        unsigned char *distanz;
        unsigned char *survivor;
        unsigned char *v_u_trellis_map;
        unsigned char h1, h2;

        //Matrixpuffer
        if ((strlen(matrix1.c_str())!=strlen(matrix2.c_str()))||(strlen(matrix2.c_str())!=strlen(matrix3.c_str()))) { ShowMessage("Bitte Matrix G auf
gleiche Zeilenlängen überprüfen."); return ""; }

        k = strlen(matrix1.c_str())-1; //z.B. G hat 6 Spalten -> k=5 innere Zustände
        laenge = strlen(eingabe.c_str())/3;
        //distanz  = (unsigned char *) malloc((laenge+1)*(1<<k)); //Eingabelänge / 3 * 2^k
        //survivor = (unsigned char *) malloc((laenge+1)*(1<<k)); //Eingabelänge / 3 * 2^k



        //MOD für Punktierung: Eingabe mit punktierten Stellen wieder anfüllen:
        unsigned char P = strlen(p1.c_str());        //z.B. P hat 6 Spalten -> P=6
        int i2=0;
        while (i2<strlen(eingabe.c_str()))
        {
          for (int i1=0; i1<P; i1++)
          {
            if      (p1.c_str()[i1]=='1') { eingabe2 = eingabe2+AnsiString(eingabe.c_str()[i2]); i2++;}
            else if (p1.c_str()[i1]=='0') { eingabe2 = eingabe2+"2"; }  // 2 markiere punktierte Stelle = 0,5
            if      (p2.c_str()[i1]=='1') { eingabe2 = eingabe2+AnsiString(eingabe.c_str()[i2]); i2++;}
            else if (p2.c_str()[i1]=='0') { eingabe2 = eingabe2+"2"; }
            if      (p3.c_str()[i1]=='1') { eingabe2 = eingabe2+AnsiString(eingabe.c_str()[i2]); i2++;}
            else if (p3.c_str()[i1]=='0') { eingabe2 = eingabe2+"2"; }
          }
        }
        laenge2 = strlen(eingabe2.c_str())/3;
        distanz  = (unsigned char *) malloc((laenge2+1)*(1<<k)); //Eingabelänge / 3 * 2^k
        survivor = (unsigned char *) malloc((laenge2+1)*(1<<k)); //Eingabelänge / 3 * 2^k



        // Trellis:

        //Generierung einer "Map", um ein verkürztes Trellis zum "Nachschlagen"
        //(direkte Auswahl) zu haben, statt ständiger Neuberechnung
        //
        //Form: sigma(neuer Wert) | sigma'(alter Wert)_1 | sigma'(alter Wert)_2 | v_1 | v_2 | u_1 | u_2
        //      (entspr. Stelle)    (implizit)             (implizit)             m bit m bit 1 bit 1 bit -> seien 1 Char (m=3)
        //mit insgesamt 2^k Einträgen
        v_u_trellis_map = (unsigned char *) malloc(1<<k);

        for (unsigned int j=0; j<(1<<(k-1)); j++) //k = 5 -> 2^5 = 0..31, sigma(neu) iterieren
        {
           //Einzelkomponenten von v generieren:
           //hier erstmal 0..15 (führende 0 -> u=0),
           /*v ausnullen*/ for (int w1=0; w1<2; w1++) for (int w2=0; w2<3; w2++) v[w1][w2]=0;
           //p=0 (entspr. Matrix*u) fällt weg, da u=0
           for (unsigned int p=1; p<=k; p++)
           {
              //v1_1:
              v[0][0] = v[0][0] ^ (matrix1.c_str()[p]-48)*(((j<<1)&(1<<(k-p)))>>(k-p)); // ^ als XOR
                                                                                        // (j<<1) liefert sigma' (für 0 rausgeschoben)
                                                                                        // &(1<<(k-p)) liefert p-tes Bit von links
                                                                                        // >>p zurückshiften
              //v1_2:
              v[0][1] = v[0][1] ^ (matrix2.c_str()[p]-48)*(((j<<1)&(1<<(k-p)))>>(k-p));
              //v1_3:
              v[0][2] = v[0][2] ^ (matrix3.c_str()[p]-48)*(((j<<1)&(1<<(k-p)))>>(k-p));
              //v2_1:
              v[1][0] = v[1][0] ^ (matrix1.c_str()[p]-48)*((((j<<1)+1)&(1<<(k-p)))>>(k-p)); // (j<<1)+1 liefert sigma' (für 1 rausgeschoben)

              //v2_2:
              v[1][1] = v[1][1] ^ (matrix2.c_str()[p]-48)*((((j<<1)+1)&(1<<(k-p)))>>(k-p));
              //v2_3:
              v[1][2] = v[1][2] ^ (matrix3.c_str()[p]-48)*((((j<<1)+1)&(1<<(k-p)))>>(k-p));
           }

           //zusammensetzen:
           v_u_trellis_map[j]            = (v[0][0]<<7) + (v[0][1]<<6) + (v[0][2]<<5) + (v[1][0]<<4) + (v[1][1]<<3) + (v[1][2]<<2); // u_1 = u_2 = 0


           //Einzelkomponenten von v generieren:
           //nun für 16..31 (führende 1 -> u=1)
           /*v ausnullen*/ for (int w1=0; w1<2; w1++) for (int w2=0; w2<3; w2++) v[w1][w2]=0;
           //p=0 (entspr. Matrix*u) behandeln, da u=1
```

```
            for (int w1=0; w1<2; w1++) { v[w1][0] = matrix1.c_str()[0]-48; v[w1][1] = matrix2.c_str()[0]-48; v[w1][2] = matrix3.c_str()[0]-48; }
            for (unsigned char p=1; p<=k; p++)
            {
               //v1_1:
               v[0][0] = v[0][0] ^ (matrix1.c_str()[p]-48)*(((j<<1)&(1<<(k-p))))>>(k-p));
               //v1_2:
               v[0][1] = v[0][1] ^ (matrix2.c_str()[p]-48)*(((j<<1)&(1<<(k-p))))>>(k-p));
               //v1_3:
               v[0][2] = v[0][2] ^ (matrix3.c_str()[p]-48)*(((j<<1)&(1<<(k-p))))>>(k-p));

               //v2_1:
               v[1][0] = v[1][0] ^ (matrix1.c_str()[p]-48)*((((j<<1)+1)&(1<<(k-p))))>>(k-p));
               //v2_2:
               v[1][1] = v[1][1] ^ (matrix2.c_str()[p]-48)*((((j<<1)+1)&(1<<(k-p))))>>(k-p));
               //v2_3:
               v[1][2] = v[1][2] ^ (matrix3.c_str()[p]-48)*((((j<<1)+1)&(1<<(k-p))))>>(k-p));
            }

            //zusammensetzen:
            v_u_trellis_map[(1<<(k-1))+j] = (v[0][0]<<7) + (v[0][1]<<6) + (v[0][2]<<5) + (v[1][0]<<4) + (v[1][1]<<3) + (v[1][2]<<2) + 3; // u_1 = u_2 = 1
         }


         // Viterbi-Algorithmus:
         //MOD für Punktierung

         //t=0 (i=0) umsetzen:
         distanz[0]=0;
         for (int w1=1; w1<(1<<k); w1++) distanz[w1]=(1<<(k+1));        for (int w1=0; w1<(1<<k); w1++) survivor[w1]=0;
            for (unsigned int i=1; i<=laenge2; i++)
         {
            //y_triplet = ((eingabe.c_str()[3*(i-1)]-48)<<2)+((eingabe.c_str()[3*(i-1)+1]-48)<<1)+(eingabe.c_str()[3*(i-1)+2]-48);
            //MOD für Punktierung:
            y_triplet = ((eingabe2.c_str()[3*(i-1)]-48)<<4)+((eingabe2.c_str()[3*(i-1)+1]-48)<<2)+(eingabe2.c_str()[3*(i-1)+2]-48);
            //für 0.5 (als 2=10 kodiert) ist dann genau das jeweils linke in dieser 2bit-Umsetzung kodiert
            //hammingdistMOD kommt immer als das Doppelte zurück (um 0.5 zu "vermeiden") -> am Ende halbieren

            for (unsigned int j=0; j<(1<<(k-1)); j++) //k = 5 -> 2^5 = 0..31
            //aus j (entpr. sigma) folgen beide Vorgänger und ihre Übergangsbewertungen eindeutig gemäß G
            {
               //hier erstmal 0..15 (führende 0 -> u=0),
               h1 = distanz[(1<<k)*(i-1)+(j<<1)]   + hammingdistMOD( (v_u_trellis_map[j]    >>5), y_triplet); // >>5 liefert erste 3 Bit
               h2 = distanz[(1<<k)*(i-1)+(j<<1)+1] + hammingdistMOD(((v_u_trellis_map[j]&28)>>2), y_triplet); // liefert zweite 3 Bit

               distanz[(1<<k)*i+j]            = min(h1, h2);
               survivor[(1<<k)*i+j]           = min_sl(h1, h2); //select, liefert 0 od. 1 je nachdem

               //nun 16..31 -> führende 1 -> u=1
               //(führende 1 abschneiden ("rausshiften"), da k nicht notwenigerweise identisch zu Bitzahl int
               h1 = distanz[(1<<k)*(i-1)+(j<<1)]   + hammingdistMOD( (v_u_trellis_map[(1<<(k-1))+j]    >>5), y_triplet); // >>5 liefert erste 3 Bit
               h2 = distanz[(1<<k)*(i-1)+(j<<1)+1] + hammingdistMOD(((v_u_trellis_map[(1<<(k-1))+j]&28)>>2), y_triplet); // liefert zweite 3 Bit

               distanz[(1<<k)*i+(1<<(k-1))+j]  = min(h1, h2);
               survivor[(1<<k)*i+(1<<(k-1))+j] = min_sl(h1, h2);
            }
         }

         //min. Abstand zur Empfangsfolge in Metrik distanz[(1<<k)*laenge+0] gespeichert -> ausgeben über in/out-value
         Metrik = distanz[(1<<k)*laenge2+0]; //enthält doppelten Wert! (im Programm halbieren)


         //Zurückverfolgung des Survivors:
         int s = 0; //Ausgangspunkt ist sigma = (0...0)
         /* //Alternative: Keine Terminierung annehmen und Distanzminimum als Startwert für Survivor-Rückverfolgung annehmen:
         int min = laenge*3;
         for (int p=0; p<(1<<k); p++) //Minimum aus letzter Stelle
         {
            ShowMessage(AnsiString((unsigned int)distanz[(1<<k)*laenge+p]));
            if (distanz[(1<<k)*laenge+p]<=min)
            {
               min = distanz[(1<<k)*laenge+p];
               s = p;
            }
         }      */

         for (int i=0; i<laenge2; i++)
         {
            reverse = reverse + AnsiString((char)((s>>(k-1))+48)); //1. Bit von sigma impliziert Eingabe u
            s = (s<<1)%(1<<k) + survivor[(1<<k)*(laenge2-i)+s]; //vorheriger Zustand, abh. von Survivor (+/- 1)
         }

         ausgabe="";
         for (int i=0; i<laenge2-k; i++) ausgabe = ausgabe + reverse[laenge2-i]; //-k -> Terminierung entfernen

         free(v_u_trellis_map);
         free(distanz);
         free(survivor);

            return ausgabe;
}

//MOD für Punktierung:
unsigned char hammingdistMOD(unsigned char s, unsigned char y)
//(nur 3-bit)
{
   unsigned char abstand;

   //0.5er gesetzt:          sonst          klassisches             * 2
   //Abstand immer 0.5                      XOR anwenden
   abstand = ((y& 2)>>1) + ((1-((y& 2)>>1))*(((y& 1)   )^((s&1)   ))<<1)
           + ((y& 8)>>3) + ((1-((y& 8)>>3))*(((y& 4)>>2)^((s&2)>>1))<<1)
           + ((y&32)>>5) + ((1-((y&32)>>5))*(((y&16)>>4)^((s&4)>>2))<<1);
```

```
   return abstand;
   //Abstand entspricht seinem doppelten, um 0.5 zu "vermeiden"
}

unsigned char min(unsigned char h1, unsigned char h2)
{
  if (h1>h2) return h2;
  else return h1;
}

unsigned char min_s1(unsigned char h1, unsigned char h2)
{
  if (h1>h2) return 1;
  else return 0;
}

String EncodeMOD(String eingabe, String matrix1, String matrix2, String matrix3, String p1, String p2, String p3)
{
          String ausgabe;
        unsigned char v[2][3];
        unsigned char k;
        int laenge;
        unsigned char *v_u_trellis_map;

        //Matrixpuffer
        if ((strlen(matrix1.c_str())!=strlen(matrix2.c_str()))||(strlen(matrix2.c_str())!=strlen(matrix3.c_str()))) { ShowMessage("Bitte Matrix G auf
gleiche Zeilenlängen überprüfen."); return ""; }

        k = strlen(matrix1.c_str())-1; //z.B. G hat 6 Spalten -> k=5 innere Zustände


        //Eingabe-/Ausgabepuffer
        for (int i=0; i<k; i++) eingabe = eingabe + "0"; //Terminierung anhängen
        laenge = strlen(eingabe.c_str());


        // Trellis:

        //Generierung einer "Map", um ein verkürztes Trellis zum "Nachschlagen"
        //(direkte Auswahl) zu haben, statt ständiger Neuberechnung
        //
        //Form: sigma(neuer Wert) | sigma'(alter Wert)_1 | sigma'(alter Wert)_2 | v_1 | v_2 | u_1 | u_2
        //      (entspr. Stelle)    (implizit)             (implizit)             m bit m bit 1 bit 1 bit -> seien 1 Char (m=3)
        //mit insgesamt 2^k Einträgen
        v_u_trellis_map = (unsigned char *) malloc(1<<k);

        for (unsigned int j=0; j<(1<<(k-1)); j++) //k = 5 -> 2^5 = 0..31, sigma(neu) iterieren
        {
           //Einzelkomponenten von v generieren:
           //hier erstmal 0..15 (führende 0 -> u=0),
           /*v ausnullen*/ for (int w1=0; w1<2; w1++) for (int w2=0; w2<3; w2++) v[w1][w2]=0;
           //p=0 (entspr. Matrix*u) fällt weg, da u=0
           for (unsigned int p=1; p<=k; p++)
           {
              //v1_1:
              v[0][0] = v[0][0] ^ (matrix1.c_str()[p]-48)*(((j<<1)&(1<<(k-p)))>>(k-p)); // ^ als XOR
                                                                                        // (j<<1) liefert sigma' (für 0 rausgeschoben)
                                                                                        // &(1<<(k-p)) liefert p-tes Bit von links
                                                                                        // >>p zurückshiften
              //v1_2:
              v[0][1] = v[0][1] ^ (matrix2.c_str()[p]-48)*(((j<<1)&(1<<(k-p)))>>(k-p));
              //v1_3:
              v[0][2] = v[0][2] ^ (matrix3.c_str()[p]-48)*(((j<<1)&(1<<(k-p)))>>(k-p));
              //v2_1:
              v[1][0] = v[1][0] ^ (matrix1.c_str()[p]-48)*((((j<<1)+1)&(1<<(k-p)))>>(k-p)); // (j<<1)+1 liefert sigma' (für 1 rausgeschoben)

              //v2_2:
              v[1][1] = v[1][1] ^ (matrix2.c_str()[p]-48)*((((j<<1)+1)&(1<<(k-p)))>>(k-p));
              //v2_3:
              v[1][2] = v[1][2] ^ (matrix3.c_str()[p]-48)*((((j<<1)+1)&(1<<(k-p)))>>(k-p));
           }

           //zusammensetzen:
           v_u_trellis_map[j]            = (v[0][0]<<7) + (v[0][1]<<6) + (v[0][2]<<5) + (v[1][0]<<4) + (v[1][1]<<3) + (v[1][2]<<2); // u_1 = u_2 = 0


           //Einzelkomponenten von v generieren:
           //nun für 16..31 (führende 1 -> u=1)
           /*v ausnullen*/ for (int w1=0; w1<2; w1++) for (int w2=0; w2<3; w2++) v[w1][w2]=0;
           //p=0 (entspr. Matrix*u) behandeln, da u=1
           for (int w1=0; w1<2; w1++) { v[w1][0] = matrix1.c_str()[0]-48; v[w1][1] = matrix2.c_str()[0]-48; v[w1][2] = matrix3.c_str()[0]-48; }
           for (unsigned char p=1; p<=k; p++)
           {
              //v1_1:
              v[0][0] = v[0][0] ^ (matrix1.c_str()[p]-48)*(((j<<1)&(1<<(k-p)))>>(k-p));
              //v1_2:
              v[0][1] = v[0][1] ^ (matrix2.c_str()[p]-48)*(((j<<1)&(1<<(k-p)))>>(k-p));
              //v1_3:
              v[0][2] = v[0][2] ^ (matrix3.c_str()[p]-48)*(((j<<1)&(1<<(k-p)))>>(k-p));

              //v2_1:
              v[1][0] = v[1][0] ^ (matrix1.c_str()[p]-48)*((((j<<1)+1)&(1<<(k-p)))>>(k-p));
              //v2_2:
              v[1][1] = v[1][1] ^ (matrix2.c_str()[p]-48)*((((j<<1)+1)&(1<<(k-p)))>>(k-p));
              //v2_3:
              v[1][2] = v[1][2] ^ (matrix3.c_str()[p]-48)*((((j<<1)+1)&(1<<(k-p)))>>(k-p));
           }

           //zusammensetzen:
           v_u_trellis_map[(1<<(k-1))+j] = (v[0][0]<<7) + (v[0][1]<<6) + (v[0][2]<<5) + (v[1][0]<<4) + (v[1][1]<<3) + (v[1][2]<<2) + 3; // u_1 = u_2 = 1
        }
```

```
        //eigentliche Enkodierung (nun sehr einfach, reines Auswählen):
        ausgabe="";
        unsigned char s = 0; unsigned char s_alt = 0;
        unsigned char vv = 0;
        for (unsigned int i=0; i<laenge; i++)
        {
           s_alt=s;
           s=(s+((eingabe.c_str()[i]-48)<<k))>>1;  //u = eingabe[i] , -48 wg. ASCII
           if ((s_alt&1)==0) vv = ( (v_u_trellis_map[s]    >>5)); // >>5 liefert erste 3 Bit
           else              vv = (((v_u_trellis_map[s]&28)>>2));

           ausgabe = ausgabe + AnsiString((char)(((vv&4)>>2)+48));  //ausgabe[i*3  ] = ((vv&4)>>2)+48;
           ausgabe = ausgabe + AnsiString((char)(((vv&2)>>1)+48));  //ausgabe[i*3+1] = ((vv&2)>>1)+48;
           ausgabe = ausgabe + AnsiString((char)(( vv&1    )+48));  //ausgabe[i*3+2] = ( vv&1    )+48;
        }



        //MOD für Punktierung: punktierten Stellen in Ausgabe löschen:
        String ausgabe2="";
        unsigned char P = strlen(p1.c_str());        //z.B. P hat 6 Spalten -> P=6
        int i2=0;
        while (i2<strlen(ausgabe.c_str()))
        {
          for (int i1=0; i1<P; i1++)
          {
            if (p1.c_str()[i1]=='1') ausgabe2 = ausgabe2+AnsiString(ausgabe.c_str()[i2]); i2++;
            if (p2.c_str()[i1]=='1') ausgabe2 = ausgabe2+AnsiString(ausgabe.c_str()[i2]); i2++;
            if (p3.c_str()[i1]=='1') ausgabe2 = ausgabe2+AnsiString(ausgabe.c_str()[i2]); i2++;
          }
        }



        free(v_u_trellis_map);

           return ausgabe2;
}
```