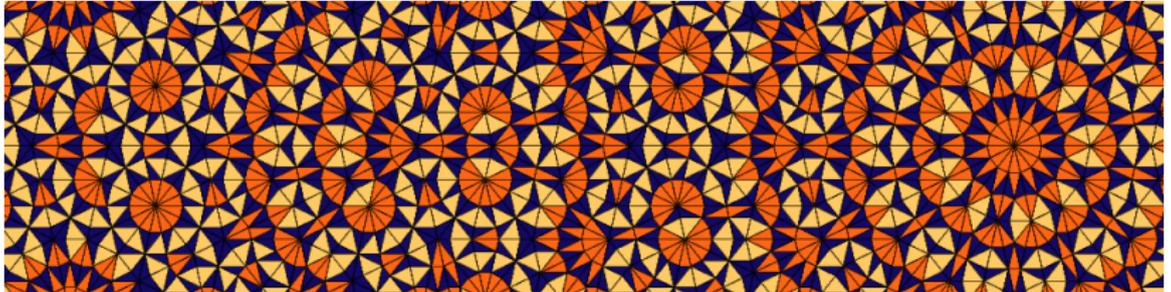


18: Algorithmen III: png und Co

Dirk Frettlöh
Technische Fakultät / Richtig Einsteigen

11.6.2015



Kompressionsalgorithmen:

Idee: Speichere 2 MB Daten in einer 1 MB Datei. Ist, je nach Daten, einfach bis sehr, sehr schwierig. Einfaches Bsp.:

00000000000000000001111111111111111100000000000011111000000000

Kniffliges Bsp.:

47238905619069345682390129012012356890317329045290239230237851

Häufig genutzte Verfahren:

- ▶ Run length encoding: (RLE) Blocklängen nutzen
- ▶ Huffman coding: häufige Zeichen mit wenig Bits (s.u.)
- ▶ LZ77, LZ78, LZW: Wiederholungen notieren (Wörterbuch anlegen, s.u.)

Bsp. RLE: String oben speichern als:

18 0en, 18 1en, 12 0en, 5 1en, 9 0en.

Also etwa: (18,0,18,1,12,0,5,1,9,0). 10 statt 62 Zahlen.

Komprimieren (also “kleiner machen”) klappt nur, wenn die Daten irgendeine Regelmäßigkeit aufweisen.

Komplett zufällige Daten (“hohe Entropie”, siehe *Informationstheorie*) lassen sich nur schwer (oder gar nicht) komprimieren.

Anwendungen

DEFLATE: in gzip, WinZIP, MacOS Archivmanager... kombiniert LZ77 und Huffman coding.

(zip ist ein Dateiformat zum Packen mehrerer Dateien, das verschiedene Kompressionsmethoden benutzt, gzip packt nur eine Datei, und immer mit DEFLATE)

- ▶ Grafik: jpeg, gif, png... auch in pdf.
- ▶ Audio: mp3, ogg, MPEG-4, ...
- ▶ Video: flv, mov, MPEG-4, ...

Hier: Details zu jpeg und png (gif ist sehr ähnlich zu png)

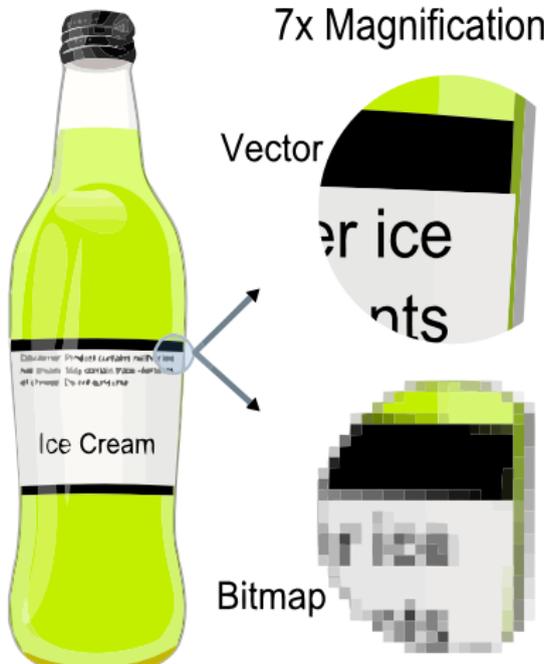
Ein naives Grafikformat wäre "Pixel 1 hat Farbe #e34d76, Pixel 2 hat Farbe #ea4d73, Pixel 3 hat Farbe #f14e69, ..."

Bekannt sein sollte: Farben werden digital als RGB (Rot-Grün-Blau) oder CMYK (cyan-magenta-yellow-black) gespeichert. Z.B. 8 bits für Rotanteil (0=00=wenig, 255=ff=knallrot), 8 bits für Grün, 8 bits für Blau. Z.B. # ff 00 00, # fd d7 4b

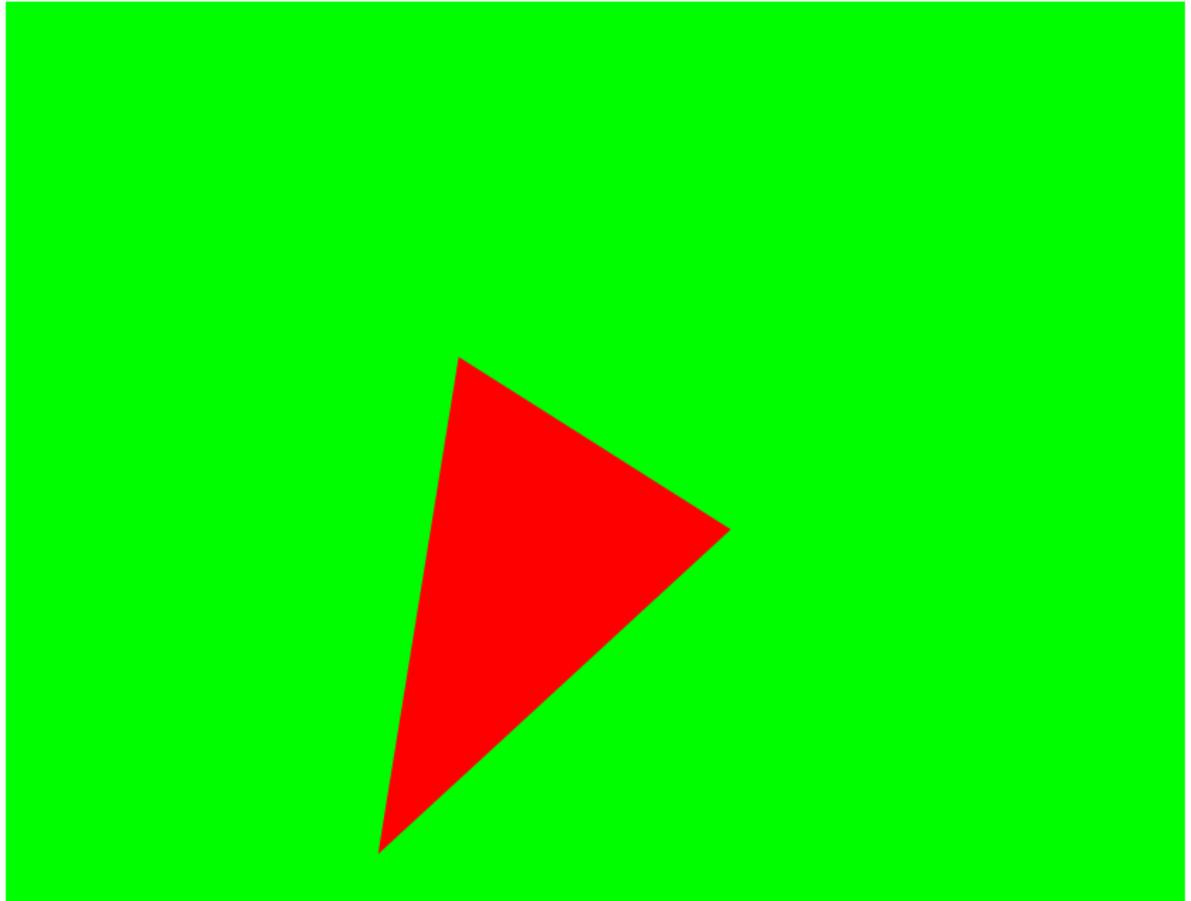
Rastergraphik vs Vektorgraphik

Rastergraphik: (engl. Bitmap): Bild besteht aus Pixeln. jpeg, png, gif, bmp, tiff...

Vektorgrafik nicht pixelbasiert. svg, ps, eps, pdf, ai, cdr, tikz....



Ein einfaches Bild, als pdf gespeichert. Reinzoomen!



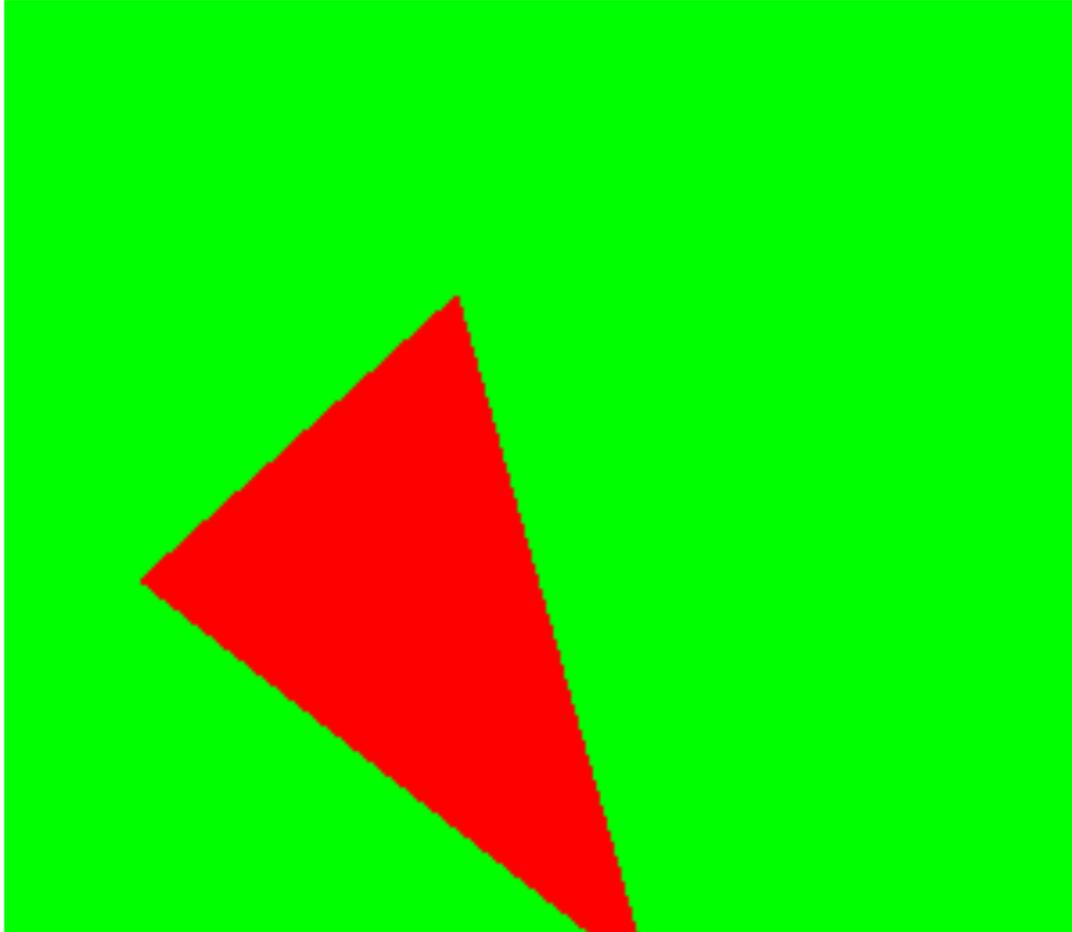
Inhalt der pdf-Datei:

```
%PDF-1.4
3 0 obj
<< /Length 4 0 R
/Filter /FlateDecode
>>
stream
dfjkhgsdfkhsdfjkjhgksdfjh
endstream
endobj
4 0 obj
101
endobj
2 0 obj
<<
/ExtGState <<
/a0 << /CA 1 /ca 1 >>
>>
>>
endobj
5 0 obj
<< /Type /Page
/Parent 1 0 R
/MediaBox [ 0 0 267.428558 221.714279 ]
/Contents 3 0 R
/Group <<
/Type /Group
/S /Transparency
/CS /DeviceRGB
>>
/Resources 2 0 R
>>
endobj

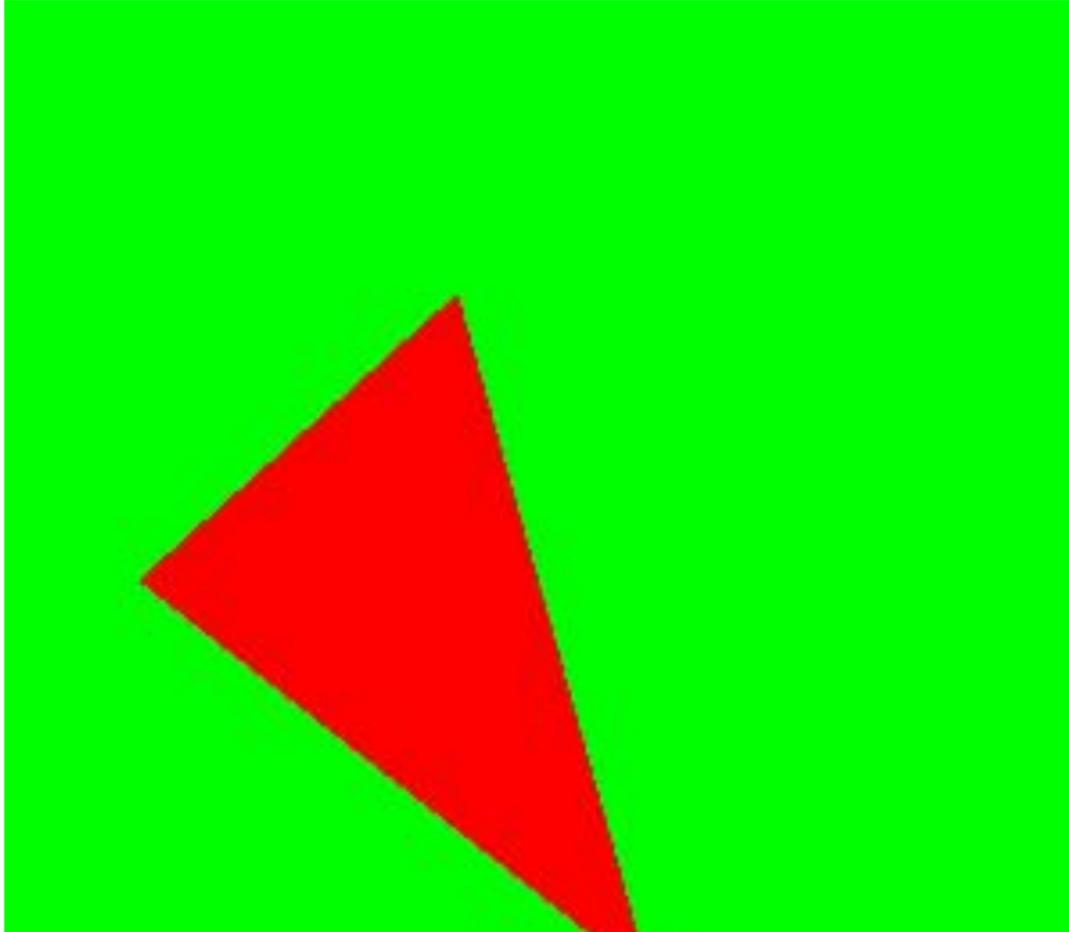
1 0 obj
<< /Type /Pages
/Kids [ 5 0 R ]
/Count 1
>>
endobj
6 0 obj
<< /Creator (cairo 1.10.2 (http://cairographics.org))
/Producer (cairo 1.10.2 (http://cairographics.org))
>>
endobj
7 0 obj
<< /Type /Catalog
/Pages 1 0 R
>>
endobj
xref
0 8
0000000000 65535 f
0000000501 00000 n
0000000215 00000 n
0000000015 00000 n
0000000193 00000 n
0000000287 00000 n
0000000566 00000 n
0000000693 00000 n
trailer
<< /Size 8
/Root 7 0 R
/Info 6 0 R
>>
startxref
745

%%EOF
```

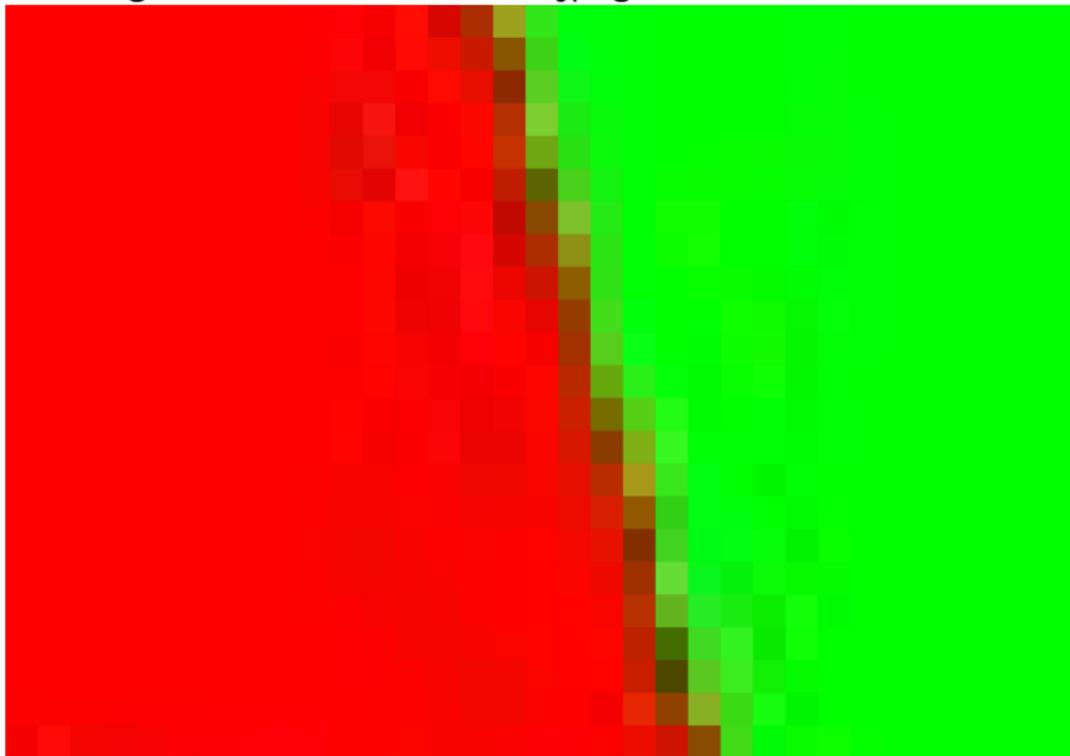
Dasselbe Bild als png gespeichert:



Dasselbe Bild als jpeg gespeichert:



Ein vergrößerter Ausschnitt des jpeg-Bilds:



Zwei Phasen beim Speichern eines Bildes:

- ▶ Filtern (Hier: rate nächsten Farbwert, speichere Differenz)
- ▶ Komprimieren (erst LZ77, dann Huffman-Coding)

1. Filtern: Durchlaufe das Bild zeilenweise Pixel für Pixel.
Rate die Farbe des aktuellen Pixels X aus den Nachbarn.

	C	B	D	
	A	X		

Vier Varianten:

An Stelle X: (A,B,C bereits bekannt)

- ▶ Rate: A,
- ▶ Rate: B,
- ▶ Rate: $\frac{1}{2}(A+B)$,
- ▶ Rate: Wert, der am nächsten an $p = A+B-C$ ist.

Speichere die Differenz zwischen geratenem und korrektem Wert.

Warum ist das klug? In Bildern oft  oder 

Änderung (Differenz) von links nach rechts z.B.: 111111111...
(*Farbverläufe* zeigen evtl. mehr Regelmäßigkeit als Farbwerte)

2. Komprimieren. Zunächst **LZ77**-Algorithmus:

Prinzip: Lesen wir einen String zum zweiten Mal, dann wird nur der Verweis gespeichert: Siehe Wort Nummer 44.

Genauer (dies ist nur ein einfach zu erklärendes Schema, es gibt etliche clevere Implementierungen):

1. Lese String von links nach rechts, Zeichen für Zeichen.
2. Solange das aktuelle Wort bereits bekannt ist: weiterlesen.
3. Ist das aktuelle Wort unbekannt, wird es in einem **Wörterbuch** gespeichert, in der Form (Nummer, Nummer-bekanntes-Wort letzter-Buchstabe). Aktuelles Wort := \emptyset , weiterlesen (weiter bei 1.)

Bsp.: Komprimiere AABABBBABAABABBBABBABB.

Erstes Wort: A. Unbekannt, also: Wort 1 = A.

A|ABABBBABAABABBBABBABB

Zweites Wort: A. Bekannt, nichts tun.

A|ABABBBABAABABBBABBABB

Drittes Wort: AB. Unbekannt, also: Wort 2 = 1B (= Wort 1-B).

A|AB|ABBBABAABABBBABBABB

Viertes Wort: A. Bekannt.

Fünftes Wort: AB, Bekannt.

Sechstes Wort ABB, unbekannt: also Wort 3 = 2B.

A|AB|ABB|BABAABABBBABBABB

Siebtes Wort: B, unbekannt: Wort 4 = $\emptyset B$ (=Wort 0-B).

A|AB|ABB|B|ABAABABBBABBABB

Achtes Wort: A. Neuntes Wort: AB.

Zehntes Wort: ABA, unbekannt: Wort 5 = 2A.

A|AB|ABB|B|ABA|ABABBBABBABB

Usw.

A|AB|ABB|B|ABA|ABAB|BB|ABBA|BB

(Das letzte Wort BB ist bekannt, speichern wir trotzdem.) Also:

1	2	3	4	5	6	7	8	9
A	AB	ABB	B	ABA	ABAB	BB	ABBA	BB
$\emptyset A$	1B	2B	$\emptyset B$	2A	5B	4B	3A	7-

Wir speichern: $\emptyset A$ als (000, 0), 1B als (001, 1), 2B als (010, 1),
 $\emptyset B$ als (000, 1), 2A als (010, 0) usw. Also:

(000, 0), (001, 1), (010, 1), (000, 1), (010, 0), (101, 1), (100, 1), (011, 0), (111, -)

Oben nur Buchstaben A,B, also nur ein bit pro Buchstabe. Mehr Buchstaben, mehr bits.

Dekodieren:

(000, 0), (001, 1), (010, 1), (000, 1), (010, 0), (101, 1), (100, 1), (011, 0), (111, -)

Wort 1: A, also A|.

Wort 2: 1B, also AB, also A|AB|. (nicht AB!)

Wort 3: 2B, also ABB, also A|AB|ABB.

Wort 4: $\emptyset B$, also B, also A|AB|ABB|B.

Wort 5: 2A, also ABA, also A|AB|ABB|B|ABA.

Wort 6: 5B, also ABAB, also A|AB|ABB|B|ABA|ABAB.

Wort 7: 4B, also BB, also A|AB|ABB|B|ABA|ABAB|BB.

Wort 8: 3A, also ABBA, also A|AB|ABB|B|ABA|ABAB|BB|ABBA.

Wort 9: 7-, also BB-, also A|AB|ABB|B|ABA|ABAB|BB|ABBA|BB.

Soviel zu LZ77. Das hier zeigt die Idee, ist einfach zu erklären und zu analysieren; reale Implementierungen sind im Detail anders.
(mit Überlappungen, Versatz speichern)

LZ77: Lempel-Ziv 1977. Nachfolger: LZ78, LZW (darauf zeitweise Patente, daher oft das freie LZ77).

Dann **Huffman coding**: Speichere häufige Symbole mit wenig Bits.

Beispiel: Nachricht: 11112233455555666666

Naiv: 20 Zeichen. 8 Bits pro Zeichen macht 160 Bits.

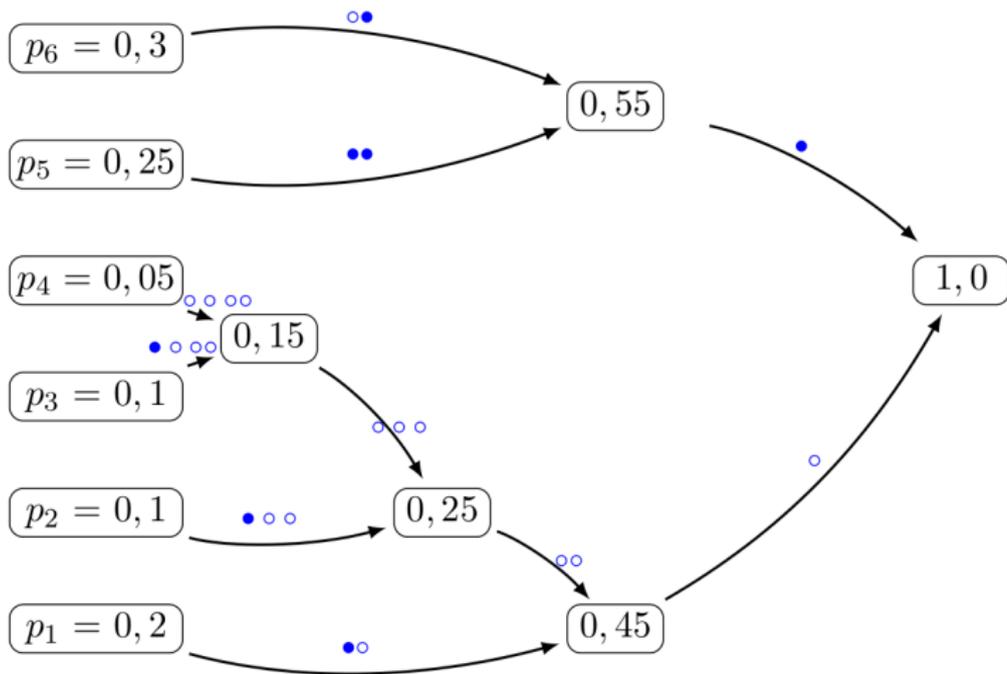
Normal: Nur 6 verschiedene Zeichen, also nur 3 Bits pro Zeichen nötig. Also $20 \cdot 3 = 60$ Bit Speicherbedarf.

Clever: Huffman-codiert:

●○|●○|●○|●○|●○○|●○○|●○○○|●○○○|○○○○|●●|●●|●●|●●|●●|○●|○●|○●|○●|○●|○●

$4 \cdot 2 + 2 \cdot 3 + 2 \cdot 4 + 1 \cdot 4 + 5 \cdot 2 + 6 \cdot 2 = 48$ Bit Speicher.

Im Schnitt 2,4 Bit pro Wort.



png vs jpg vs gif vs ...

png speichert verlustfrei (**lossless**, keine Information geht verloren).

Es kann auch stärker komprimieren, wenn man verlustbehaftetes (**lossy**) Komprimieren vorwählt: Dann wird die Zahl der Farben reduziert.

Die Leistung (Komprimierungsgrad) lässt sich daher schwierig vergleichen. **jpeg** ist lossy, aber klein, insbesondere bei Fotos. **png** ist verlustfrei und klein bei wenigen Farben (s.o.), aber groß bei vielen Farben (Fotos!)

png to gif: png kann 2^{24} Farben, gif nur 2^8 . Daher wird beim Konvertieren die Datei evtl. kleiner, aber lossy!
Dafür kann gif Animationen.

Andere Formate wie **tiff** enthalten viel mehr Informationen (sehr viele Farben, mehrere Ebenen ("layer"), Transparenz, cmyk, Metadaten...), sind dafür größer.

cmyk: Cyan-Magenta-Yellow-blackK: braucht der Drucker.
Schwarz in RGB: 000000 (keine Farbe, also Bildschirm schwarz)
Aber: keine Farbe auf dem Drucker = weiß.

[Dann gezeigt: DMV-Mitteilungen 1-2014, insbesondere:
D.Kriesel: Traue keinem Scan, den du nicht selbst gefälscht hast]