Dozent: Dr.-Ing The Anh Vuong

Seminar: “Aktuelle Themen der Datenkompression”

Graphische Daten Verarbeitung, Informatik Institut

Johann Wolfgang Goethe Universität, Frankfurt am Main

Projekttitel **Run Length Codierung**

Projektautoren:

Felix Landsiedel (5285778),

Stefan Chalupka (6037666),

Marcel Brück (6192721)

Run Length Codierung

Gliederung

- Einleitung - Datenkompression
- Grundlagen des Run-Length Encoding
- Anwendungsgebiete
- Ansätze zur Optimierung
- Exkurs: Move-to-Front Coding
- Programmvorschau
- Resümee
- Literaturverzeichnis

Einleitung – Datenkompression

Die Datenkompression (engl. data compression) auch Datenkomprimierung genannt, ist ein Vorgang, bei dem die Menge digitaler Daten verdichtet oder reduziert wird.

Grundsätzlich wird bei einer Datenkompression versucht, redundante Informationen zu entfernen.

Diesen Vorgang bezeichnet man als Kompression oder Komprimierung. Die Umkehrung bezeichnet man als Dekompression oder Dekomprimierung.

Wenn aus den komprimierten Daten wieder alle Originaldaten gewonnen werden können, spricht man von verlustfreier Kompression, verlustfreier Kodierung oder auch Redundanzreduktion. Dies ist wichtig für die spätere Ausführbarkeit Programmdateien notwendig.

Zu den Verlustfreien Kodierungen gehören die Wörterbuchmethode, die Entropiekodierung sowie das Run-Length Encoding.

Run-Length Encoding wird bei verschiedenen Grafikdateiformaten wie dem Tagged Image File Format (TIFF), dem Bitmap- und dem TGA-Dateiformat eingesetzt, aber auch bei Faxübertragungen.

Grundlagen des Run-Length Encoding

Die Lauflängenkodierung (engl. run-length encoding, kurz RLE), ist ein einfacher verlustfreier Kompressionsalgorithmus. Man zählt ihn zu der Klasse der Entropiekodierer, der er gut geeignet ist, längere Wiederholungen von Symbolen zu komprimieren und somit Redundanzen beseitigt.

Innerhalb einer Zeichenfolge werden beim RLE Zeichenwiederholungen derart gespeichert, dass nur die Länge der Wiederholung bis zum nächsten Zeichen, nicht aber jedes wiederholte Zeichen einzeln gespeichert wird.

Somit würde eine Zeichenfolge der Form

- XXXXXYYYZZZZ
  - komprimiert zu
- X5Y3Z4

Zusätzlich eignet er sich sehr gut um einfache Grafiken mit wenigen Farben und größeren Flächen zu kodieren. Das Problem ist jedoch, dass ein Farbbild mit steigendem Detailumfang ungeeigneter für die RLE-Codierung wird.

Weiterhin ist durch eine solche Repräsentation von Zeichenfolgen die Handhabung kompliziert, da es in einem einmal abgespeicherten Array nicht mehr einfach ist, Änderungen an den Daten vorzunehmen. Dies liegt daran, dass in vielen Fällen kein „random access“ auf einzelne Elemente in der Mitte der Zeichenfolge mehr möglich ist – das Array kann nur sequentiell vom Anfang an gelesen werden.

Um die Effizienz des Verfahrens zu erhöhen, kann der Algorithmus eine Wegoptimierung vornehmen - durch die Wahl des effektivsten Laufwegs, der Zeilensequentiell, beziehungsweise Spaltensequentiell sein kann, oder das ganze Bild in einer mäandernden Bewegung erfassen kann.

Run-Length Encoding zur Textkompression

Zu den zunächst offensichtlichsten Anwendungen des Run-Length-Encoding gehört die Kompression von Fließtextdateien. Bei näherer Betrachtung fällt allerdings sofort auf, dass die Lauflängenkodierung von Texten schnell zu mannigfaltigen Problemen führen kann, die im folgenden Abschnitt besprochen werden.

Eines dieser Probleme ist die Eindeutigkeit, mit der ein komprimierter Text später vom Decoder interpretiert werden kann. Wenn mit der oben genannten Methode beispielsweise ein Text verarbeitet werden soll, der neben den lateinischen Buchstaben Zahlen beinhaltet, hat der Dekoder keine Möglichkeit zu unterscheiden, ob die gerade gelesene Ziffer eine Mengen-, Zeit- beziehungsweise Datumsangabe oder die Anzahl der sich wiederholenden Buchstaben angibt.

Beispielsweise würde der Satz

„In 4 von 5 Kongressstädten wird Geschirrreiniger in 10 verschiedenen Kunststoffflaschen verkauft.“

zu

In 4 von 5 Kongres3tädten wird Geschir3einiger in 10 verschiedenen Kunststof3laschen verkauft.

zusammengefasst. Bei der Dekompression würde das allerdings als

„Innnn vonnnnn Kongressstädten wird Geschirrreiniger innnnnnnnnn verschiedenen Kunststoffflaschen verkauft.“

interpretiert. Das bedeutet, dass zunächst ein neutrales Symbol, von dem man mit Sicherheit davon ausgehen kann, dass es in einem Text nicht vorkommt, verwendet werden muss, um den Beginn einer neuen Folge von Zeichenwiederholungen zu signalisieren.

Auf diesen Token muss sich allerdings von Fall zu Fall festgelegt werden, was je nach vorliegendem Text eine Veränderung des Programmcodes und damit zusätzlichen Aufwand bedeuten kann. Wird als solches Symbol nun beispielsweise das Zeichens **~** verwendet, würde der obige Text zu

In 4 von 5 Kongres~3tädten wird Geschir~3einiger in 10 verschiedenen Kunststof~3laschen verkauft.

Dieses Format ermöglicht zwar eine korrekte Reinterpretation des Textes, allerdings wird mit dem eingefügten Zeichen nun so viel zusätzlicher Platz gebraucht, dass durch das Run-Length Encoding kein Speicher mehr eingespart werden kann. Da es in der deutschen Sprache im engeren Sinne keine Worte mit mehr als 4 gleichen aufeinanderfolgenden Buchstaben gibt, lässt sich feststellen, dass sich die einzigen Möglichkeiten, eine Textdatei mittels Run-Length Encoding tatsächlich Speichereffizienter repräsentieren zu können auf die Komprimierung im Text vorkommender Absätze – längere Runs aufeinanderfolgender Leerzeichen - oder großen Zahlen beispielsweise im Quartalsbericht eines Unternehmens beschränken.

So ließe sich

„Ich hoffe es geht dir gut.

Liebe Grüße,

Martin“

durch Run-Length Encoding immerhin zu

Ich hof2e es geht dir gut. 48

Liebe Grüße, 59

Martin

von 152 auf nur 51 Zeichen verkürzen, was eine Speichereinsparung von beinahe einem Drittel bedeuten würde – vorausgesetzt ein Absatz bestünde tatsächlich aus fortlaufenden Leerzeichen.

Wie sich aus dem Beispiel des Absatzes, also einem großen Block gleichförmiger Information bereits erahnen lässt, arbeitet Run-Length Encoding sehr effektiv auf Formaten, die nur Binärdaten enthalten.

Run-Length Encoding zur Bildkompression

Wie sich im folgenden Abschnitt herausstellen wird, ist das RLE-Verfahren ein offensichtlicher Kandidat für die Kompression digitaler Bilddateien.

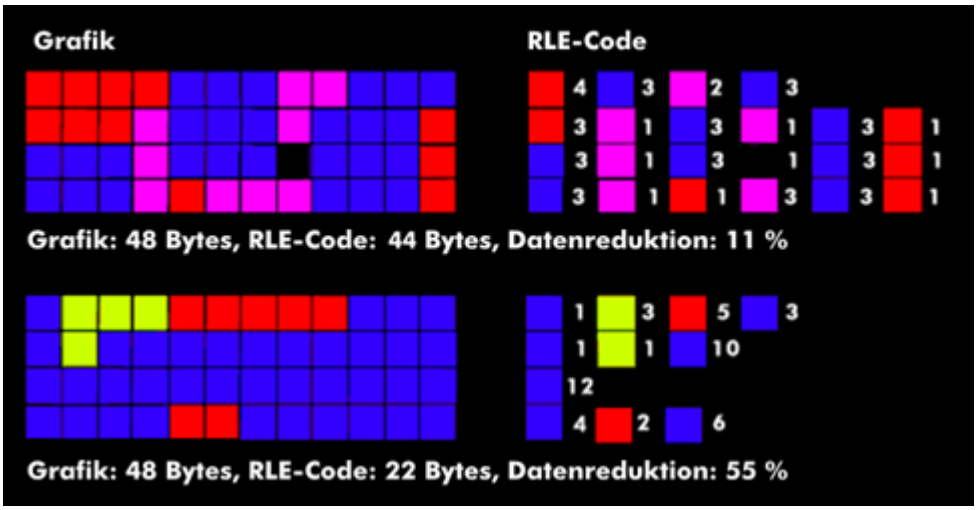
Run-Length Encoding ist verlustfrei und kann aufgrund seiner Funktionsweise keinerlei Bildartefakte – Abweichungen vom Original, kleine Bildfehler und Pixelcluster - in komprimierten, beziehungsweise später dekodierten Images erzeugen und zeigt auf dem Gebiet der Bildkompression im Schnitt auch eine deutlich bessere Performance als bei Fließtextdateien.

https://www.researchgate.net/profile/Tamer_Rabie/publication/275641512/figure/fig1/AS:280869420978176@1443975896049/JPEG-compressed-color-image-showing-blocking-artifacts-in-the-right-image-portion.png

Eine digitale Bilddatei besteht aus aneinandergereihten Pixeln, wobei jeder Pixel die Information aus einem Bit oder aus mehreren Bits enthalten kann, wobei ein Bit beispielsweise angeben kann, ob der betreffende Bildpunkt schwarz beziehungsweise weiß ist, während mit mehreren Bits die Farbe eines Pixels auf einer Farbskala kodiert werden kann.

Folgend nehmen wir an, dass alle Pixel, aus denen ein digitales Bild besteht in einem Array gespeichert sind – dieses Bitmap-Array ist der Input-Stream des Bildes für den Kompressionsalgorithmus und enthält den Pixel in der oberen linken Ecke als erstes Element und den Pixel in der unteren rechten Ecke als letztes Element.

Dass Run-Length Encoding sich prinzipiell gut zur Kompression digitaler Bilddateien eignet ist der Tatsache geschuldet, dass es in Bildern nicht unüblich ist, dass benachbarte Pixel dieselbe Farbe haben - ist auf einem Foto beispielsweise der Himmel zu sehen, lassen sich lange Pixelfolgen der Bitmaps mit derselben blauen Farbe zusammenfassen, da nur die Anzahl der aufeinander folgenden blauen Pixel gleicher Farbe in den Output-Stream geschrieben werden müssen.



https://www.itwissen.info/lex-images/rle-kompression-dot-beispiele-mit-geringer-und-hoeherer-effizienz.png

Da die Bilddatei in dieser einfachen Form des Run-Length Encoding in Zeilen vom linken zum rechten Rand des Bildes gelesen wird, lässt sich die durch Komprimierung eingesparte redundante Information wie folgt approximieren:

- Eine Fläche gleicher Farbe bestehe aus insgesamt F Pixeln.
- Die Fläche habe einen Umfang von U Pixeln
  - Dann durchkreuzen im Mittel etwa $\frac{U}{2}$ Zeilen (die keine gemeinsamen Pixel haben) die oben genannte gleichfarbige Fläche, die dafür jeweils nur einen Zähler in den Output-Stream schreiben.
- Daraus folgt, dass die Kompressionsrate ungefähr $\frac{(\frac{U}{2})}{F}$ beträgt.

Offensichtlich ist auch hier, dass die Performance der Lauflängenkodierung abnimmt, je detailreicher die übergebene Bilddatei ist.

Wird die Lauflängenkodierung für nicht-binäre Bilddateien verwendet, die also Farbe oder Graustufen beinhalten, wird jede Folge von Pixeln gleicher Intensität als Tupel (Lauflänge, Farbwert) kodiert, wobei für die Lauflänge üblicherweise mit einem Byte gerechnet wird, was bedeutet, dass Pixelfolgen mit einer Länge von bis zu 255 Pixeln berücksichtigt werden können. Die Farbwerte bestehen

folgerichtig aus beispielsweise zwischen 4 und 8 Bits, abhängig von den im Bild vorhandenen Farbwerten.

Beispiel einer Graustufen-Bitmap mit einer Farbtiefe von 8 Bit

10, 10, 10, 10, 10, 35, 5, 5, 5, 5, 5, 5, 42, 67, 87, 87, 87, 1, …

würde komprimiert zu

5, 10, 35, 6, 5, 42, 67, 3, 87, 1, …,

wobei

- **Rot** die Lauflänge und
- **Grau** den Farbwert kodiert.

Auch bei diesem Verfahren fällt dem aufmerksamen Betrachter schnell das grundlegende Problem ins Auge: Der farbenblinde Algorithmus hat keine Möglichkeit zu erkennen, welche der Ziffern eine Lauflänge und welche einen Graustufenwert repräsentiert.

Eine einfache Lösung des Problems besteht darin, die Farbtreue des kodierten Bildes minimal zu reduzieren und die 256 verschieden Abstufungen der verfügbaren Grautöne auf 255 zu verringern. Der Wert 256 wird nun zu einem Token, der dem Algorithmus den Beginn einer neuen Folge von Pixeln identischen Farbwertes signalisiert. Unser Beispiel

10, 10, 10, 10, 10, 35, 5, 5, 5, 5, 5, 5, 42, 67, 87, 87, 87, 1, …

wird dann wie folgt kodiert:

256, 5, 10, 35, 256, 6, 5, 42, 67, 256, 3, 87, 1, …

Eine weitere verbreitete Möglichkeit der Unterscheidung zwischen Farbwert und Lauflänge ist wieder die Reservierung eines Bits pro Byte um anzuzeigen, worum es sich bei der folgenden Ziffer handelt. Diesmal sind diese „Vorzeichenbits“ allerdings in Achtergruppen zusammengefasst und führen im Output immer ihre zugehörigen acht Bytes an.

10, 10, 10, 10, 10, 35, 5, 5, 5, 5, 5, 5, 42, 67, 87, 87, 87, 1, 1, 1, 9, 8, 1, …

ergäbe nach diesem Prinzip:

10010001, 5, 10, 35, 6, 5, 42, 67, 3, 01000…, 87, 3, 1, 9, 8, 1, …

Die Gesamtgröße der zusätzlich benötigten Bytes beträgt ein Achtel des Output-Streams, da pro Byte ein Bit für die Fallunterscheidung zwischen Lauflängenzähler und Farbwert benötigt wird, was den anfallenden Speicherbedarf folglich um 12.5% erhöht.

Eine letzte Möglichkeit ist, ein negatives Bit mit Wert **-M** anzulegen, wobei **M** die Länge des darauffolgenden Laufs *unterschiedlicher* Graustufenwerte beträgt.

10, 10, 10, 10, 10, 35, 37, 1, 9, 12, 42, 67, 87, 87, 87, 1, 9, 8, 4, …

sähe wie folgt aus:

5, 10, -7, 35, 37, 1, 9, 12, 42, 67, 3, 87, -4, 1, 9, 8, 4, …

Worst-Case in diesem Fall ist eine Pixelfolge der Form **(p1, p2, p2)**, die sich vom Anfang des Bitmaps bis zu dessen Ende wiederholt → **(-1, p1, 2, p2)**. Benötigt jedes Pixel ein Byte, werden statt 3 nun 4 Bytes benötigt. Werden allerdings pro Pixel beispielsweise 3 Bytes benötigt, wird die ganze Folge von 3 Pixeln (9 Byte) auf nur **1 + 3 + 1 + 3 = 8** Bytes komprimiert.

Weiterhin gilt es, bei der Bildkompression noch folgendes zu beachten:

- Da eine kodierte Lauflänge niemals kleiner als 1 sein kann, macht es durchaus Sinn, jede Aufeinanderfolge gleicher Bits mit Länge **M** als **M-1** zu speichern. So ist sichergestellt, dass die maximale Anzahl einer kodierten Lauflänge 256 beträgt.

  - **(3, 36)** bedeutet in dem Fall, das **4** Pixel der Intensität **36** aufeinander folgen.

- Bei digitalen Farbbildern ist es üblich, jeden Pixel in drei Bytes zu speichern, wobei jedes Byte jeweils die Intensität des Rot-, Grün- sowie Blauwertes angibt. In diesem Fall sollten Runs jeder Farbe separat komprimiert werden, eine Pixelfolge der Form

  (162, 82, 91), (162, 83, 91), (162, 83, 90), (162, 83, 90)

  sollte also möglichst in

  (162, 162, 162, 162), (82, 83, 83, 83), (91, 91, 90, 90)

aufgeteilt werden, wobei jede Sequenz einzeln mittels Run-Length-Encoding komprimiert wird. Da es offensichtlich deutlich wahrscheinlicher ist, dass in zwei aufeinander folgenden Pixeln die Rot-, Grün- beziehungsweise Blauwerte übereinstimmen, als dass der Rotwert eines Pixels auch mit dem Grünwert ein und desselben Pixels übereinstimmt, erhöht sich bei dieser Praxis die Wahrscheinlichkeit, lange Wiederholungen der gleichen Farbintensität zu erhalten.

Hier wird auch ersichtlich, dass jede Methode, ein Schwarz-Weiß Bild mit Graustufen mittels Run-Length Encoding zu komprimieren, im Prinzip auch für Farbbilder funktioniert.

Alternative RLE-Verfahren zur Maximierung (1) der Kompressionsrate, oder (2) der Verarbeitungsgeschwindigkeit

- Anstatt jeden neu beginnenden Run mit einem Counter zu versehen, wird erst mit der zweiten Zeichenwiederholung mitgezählt - nützlich, wenn zwischen vielen längeren Runs auch einzelne Zeichen ohne Wiederholung auftreten

  - AAAAAAACBBBBCDDE wird zu AA5-C-BB3-C-DD-E
  - Dadurch werden weniger redundante Counter benötigt

- Anstatt jedem Counter nur die Länge des jeweiligen Runs zu geben, speichere die Position des Runs im Komprimierten Array

  - AAAAAAACBBBBCDDE wird zu A1-C8-B9-C13-D14-E16
  - Keine Möglichkeit, unnötige Counter zu vermeiden, dafür ist „random access“ auf alle Elemente mittels binärer Suche möglich

- Um Vektorisierung zu ermöglichen kann die Blocklänge Vorgegeben werden, beispielsweise die Blocklänge k=2.

  - AAAAAAACBBBBCDDE wird zu 3A-1AC-2B-1CD-1DE
  - Hier fallen offensichtlich unnötig viele Runs an, dafür wird die spätere Verarbeitung schneller

Insgesamt ist ersichtlich, dass das optimale RLE-Format von der Beschaffenheit der zu komprimierenden Daten und der erdachten Verwendung abhängt, da es immer einen Trade-Off zwischen Kompressionsgüte und der Geschwindigkeit gibt, mit der die vorliegenden Daten verarbeitet werden können.

Exkurs: Move-to-Front Coding

Move-to-Front Coding ist ein Konzept, dass eigentlich aus der Burrows-Wheeler-Transformation stammt. Der Algorithmus eignet sich allerdings auch, um das Run-Length Encoding sinnvoll zu erweitern und – je nach Beschaffenheit des vorliegenden Datensatzes – speichereffizienter zu machen. Die zugrundeliegende Idee wird im folgenden Absatz vorgestellt.

Man stelle sich die vorliegenden Pixel im Bitmap als Alphabet **A** vor. Wird das Pixelarray nun sequentiell eingelesen, soll das Alphabet diejenigen Zeichen, also in unserem Fall beispielsweise die Werte der jeweiligen Farbintensität, die gehäuft im Input-Stream vorkommen an den Anfang der Liste stellen.

Sei unser Alphabet

A = (b, 3, e, 92, 5)

und das nächste zu kodierende Zeichen im Input sei **e**, dann wird **e** mit einer 2 kodiert, da es im Alphabet von zwei weiteren Zeichen angeführt wird.

In der einfachsten Variante des Move-to-Front Coding würde das **e** nun an den Anfang des Alphabets rücken

→ A = (e, b, 3, 92, 5).

Die Anwendung des Move-to-Front Coding entspringt der Idee, dass ein Zeichen, wenn es einmal gelesen wurde, vermutlich in kurzer Zeit viele weitere Male gelesen wird und, zumindest für eine gewisse Zeit ein häufig gelesenes Symbol bleibt. Diese Eigenheit des Move-to-Front Coding nennt sich *lokal adaptiv,* da sich das Alphabet in unserem Fall an die relative Dichte ähnlicher Pixel in verschiedenen Bereichen des Bildes anpasst. Wenn also tatsächlich viele identische Pixel in zusammenhängenden Bereichen des Bitmap-Arrays auftauchen, wird das Move-to-Front Coding gute Ergebnisse erzielen.

Um die Funktionsweise zu illustrieren folgen zwei Beispiele an unserem bereits definierten Alphabet

Sei unser Input

***i* = [e, b, e, 3, 3, 3, b, 5, 5, 5, 92, 92, 92, 5].**

Wird Move-to-Front Coding angewendet, erhalten wir den Output

***o1* = [2, 1, 1, 2, 0, 0, 2, 4, 0, 0, 4, 0, 0, 1]** siehe 1.1

Wird ***i*** ohne Move-to-Front Coding angewendet, erhalten wir den Output

o2 **= [2, 0, 2, 1, 1, 1, 0, 4, 4, 4, 3, 3, 3, 4]** siehe 1.2

e | b, 3, e, 92, 5 | 2	e | b, 3, e, 92, 5 | 2
b | e, b, 3, 92, 5 | 1	b | b, 3, e, 92, 5 | 0
e | b, e, 3, 92, 5 | 1	e | b, 3, e, 92, 5 | 2
3 | e, b, 3, 92, 5 | 2	3 | b, 3, e, 92, 5 | 1
3 | 3, e, b, 92, 5 | 0	3 | b, 3, e, 92, 5 | 1
3 | 3, e, b, 92, 5 | 0	3 | b, 3, e, 92, 5 | 1
b | 3, e, b, 92, 5 | 2	b | b, 3, e, 92, 5 | 0
5 | b, 3, e, 92, 5 | 4	5 | b, 3, e, 92, 5 | 4
5 | 5, b, 3, e, 92 | 0	5 | b, 3, e, 92, 5 | 4
5 | 5, b, 3, e, 92 | 0	5 | b, 3, e, 92, 5 | 4
92 | 5, b, 3, e, 92 | 4	92 | b, 3, e, 92, 5 | 3
92 | 92, 5, b, 3, e | 0	92 | b, 3, e, 92, 5 | 3
92 | 92, 5, b, 3, e | 0	92 | b, 3, e, 92, 5 | 3
5 | 92, 5, b, 3, e | 1	5 | b, 3, e, 92, 5 | 4

Tabelle 1.1 Tabelle 1.2

Ohne Move-to-Front Coding beträgt die Größe der durchschnittlichen Kodierung 2.29, während es mit MtF nur 1.21, was einen Unterschied von 1.08 beziehungsweise 47,16% ausmacht. In dem obigen Beispiel ist zu sehen, dass Move-to-Front gute Ergebnisse liefert, wenn die vorkommenden Daten dicht aneinander gleiche Werte enthalten. Das nachfolgende Beispiel soll verdeutlichen, dass Move-to-Front die Performance allerdings verschlechtert, wenn beispielsweise ein zu kodierendes Bild aus zufällig angeordneten Farbpixeln besteht. **A = (b, 3, e, 92, 5)**

Sei unser Input jetzt

i **= [e, b, 92, 3, e, 5, b, 92, 3, e, 5, b, 92, 3].**

Wird Move-to-Front Coding angewendet, erhalten wir den Output

o1 **= [2, 1, 3, 3, 3, 4, 4, 4, 4, 4, 4, 4, 4, 4]** siehe 1.1

Wird ***i*** ohne Move-to-Front Coding angewendet, erhalten wir den Output

o2 **= [2, 0, 3, 1, 2, 4, 0, 3, 1, 2, 4, 0, 3, 1]** siehe 1.2

e | b, 3, e, 92, 5 | 2	e | b, 3, e, 92, 5 | 2
b | e, b, 3, 92, 5 | 1	b | b, 3, e, 92, 5 | 0
92 | b, e, 3, 92, 5 | 3	92 | b, 3, e, 92, 5 | 3
3 | 92, b, e, 3, 5 | 3	3 | b, 3, e, 92, 5 | 1
e | 3, 92, b, e, 5 | 3	e | b, 3, e, 92, 5 | 2
5 | e, 3, 92, b, 5 | 4	5 | b, 3, e, 92, 5 | 4
b | 5, e, 3, 92, b | 4	b | b, 3, e, 92, 5 | 0
92 | b, 5, e, 3, 92 | 4	92 | b, 3, e, 92, 5 | 3
3 | 92, b, 5, e, 3 | 4	3 | b, 3, e, 92, 5 | 1
e | 3, 92, b, 5, e | 4	e | b, 3, e, 92, 5 | 2
5 | e, 3, 92, b, 5 | 4	5 | b, 3, e, 92, 5 | 4
b | 5, e, 3, 92, b | 4	b | b, 3, e, 92, 5 | 0
92 | b, 5, e, 3, 92 | 4	92 | b, 3, e, 92, 5 | 3
3 | 92, b, 5, e, 3 | 4	3 | b, 3, e, 92, 5 | 1

Tabelle 2.1 Tabelle 2.2

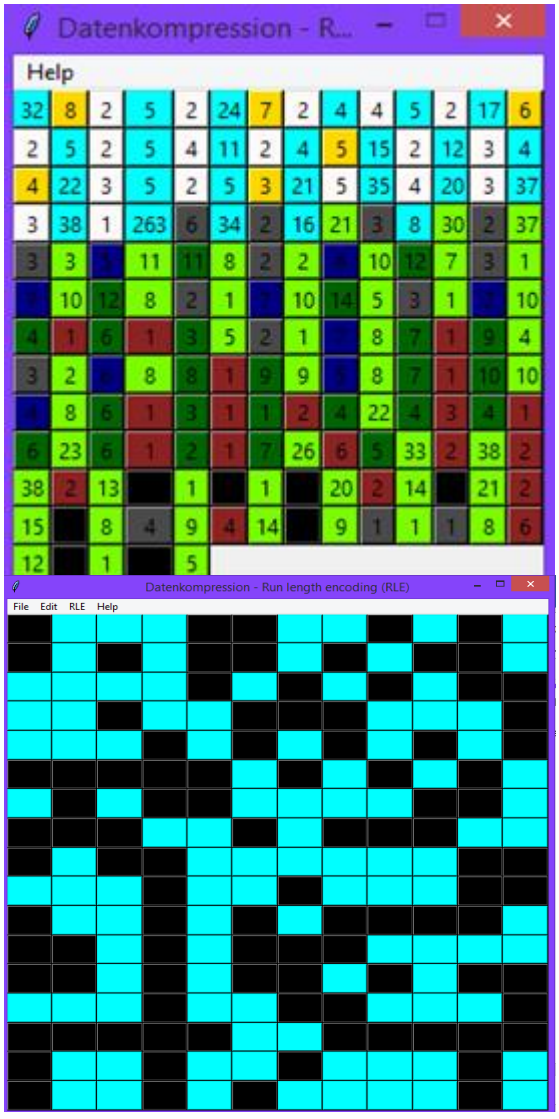
Ohne Move-to-Front Coding beträgt die Größe der durchschnittlichen Kodierung hier nur 1.86, während es mit „MtF“ bereits 3.43 sind, was einen Unterschied von 1.57 beziehungsweise 45,77% ausmacht.

Programmvorschau

Auf der folgenden Seite lassen sich die in der bisherigen Ausarbeitung gewonnenen Kenntnisse über die Funktionsweise, Stärken und Schwächen des Run-Length Encoding anhand des von der Seminargruppe entworfenen Programms überprüfen und an Beispielbildern beobachten.

Nach allen Erkenntnissen, die wir bereits über das RLE-Verfahren erlangt haben, müsste sich Bild 1 bestens mit unserem Algorithmus komprimieren

Unser erster Verdacht wird durch Bild 2 – das fertig komprimierte Pixelarray – bestätigt. Es konnten viele Wiederholungen von 20, 30 teilweise sogar von weit über 200 gleichfarbigen, redundanten Pixeln durch die Komprimierung mit unserem Run-Length Encoding Programm eingespart werden.



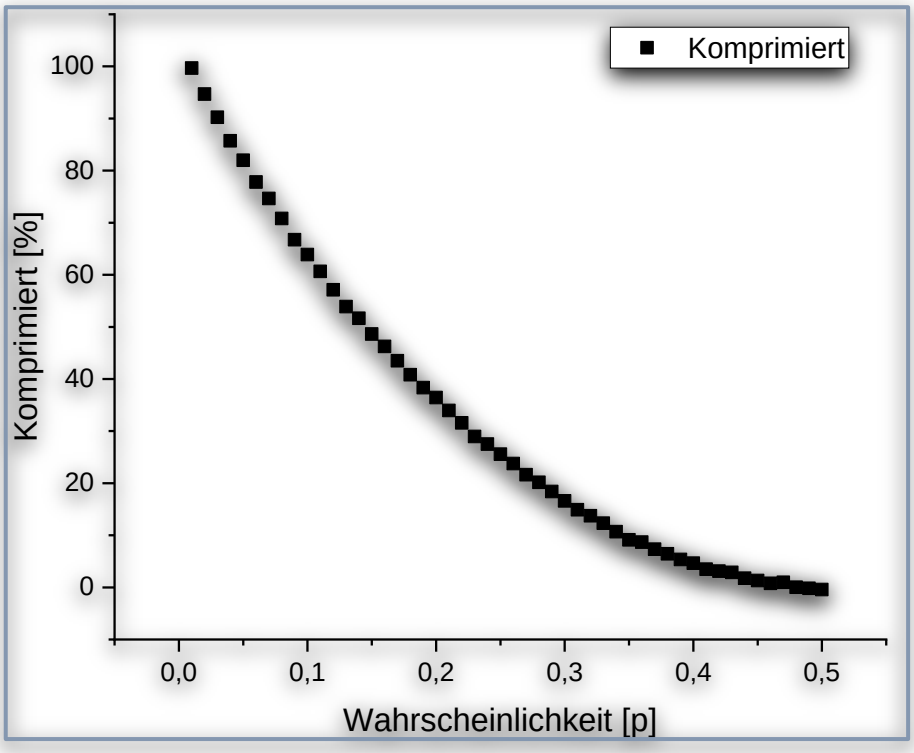
Diese Kompression ergibt eine Gesamteinsparung von 75,69% bei diesem Bild.

Das Programm kann auch mit nur zwei Farben gestartet werden statt so vielen um die Zeichenmenge zu reduzieren, wobei jede Farbe je nach Belieben eine unterschiedliche Wahrscheinlichkeit p bzw. q = 1-p hat. Mit steigender Wahrscheinlichkeit für A steigt auch die durchschnittliche Länge der Runs. Das Verhalten kann durch mehrmaliges starten des Programms mit angepassten Listen nachvollzogen werden.

Nun wird der Versuch mit 50 verschiedenen Eingangsdaten (Wahrscheinlichkeiten) gestartet, also mit

- 50A : 50B
- 51A : 49B
- ...
- 99A : 1B
- 100A: 0B

Trägt man diese Daten in ein Analyseprogramm wie Origin graphisch gegeneinander auf, erhält man den Graphen aus der Abbildung rechts. Es ist deutlich zu erkennen, dass mit sinkender Wahrscheinlichkeit von B (p) und damit steigender Wahscheinlichkeit von A die Kompression deutlich zunimmt. Dies lässt sich gut graphisch mit dem Programm (DemonstrationsKit) nachvollziehen wenn man ein bisschen mit dem Parametern herumspielt.



Es kann gut demonstriert werden wie mit steigender Wahrscheinlichkeit eines Zeichens die durchschnittliche Länge der Runs zunimmt und dadurch die Kompression verbessert wird.

Resümee

Das Verfahren der Run-Length Kodierung erzielt insgesamt bei Binärdaten außerordentlich gute Ergebnisse.

Sofern keine Daten mit niedriger Dichte gleichartiger Zeichen zu komprimieren sind spricht die vergleichsweise niedrige benötigte Rechenleistung in Verbindung mit der erreichten Kompressionsgüte sowie nicht zuletzt der Einfachheit des zu implementierenden Algorithmus für eine Verwendung des Run-Length Encoding, sofern man die gewählte Kodierungsvariante mit der (erwarteten) Beschaffenheit des Datensatzes abstimmt.

Den je nach Beschaffenheit der Daten kann RLE sehr effektiv werden und den Speicherplatzbedarf der Runs in den Daten logarithmisch verkleinern.

Von der Kompression komplexer Dateiformate wie Digitalfotos mit hoher Farbvielfalt und vielen graduellen Farbübergängen ist daher grundsätzlich abzuraten.

Man sollte sich also generell auf die Anwendung des Verfahrens auf Rastergrafiken, Piktogramme, Fax-Nachrichten oder besonders Aufgrund der Verlustfreiheit der Lauflängenkodierung auch Skizzen oder technische Zeichnungen beschränken.

Literaturverzeichnis

https://www.youtube.com/watch?v=dG92dd4iqDg

https://www.itwissen.info/RLE-run-length-encoding-Lauflaengencodierung.html

http://www.imn.htwk-leipzig.de/~medocpro/buecher/sedge1/k22t2.html

https://lemire.me/blog/2009/11/24/run-length-encoding-part-i/

https://www.prepressure.com/library/compression-algorithm/rle

http://www.stoimen.com/blog/2012/01/09/computer-algorithms-data-compression-with-run-length-encoding/

https://www.uobabylon.edu.iq/eprints/pubdoc_11_13302_25.doc

https://kogs-www.informatik.uni-hamburg.de/~neumann/Signalverarbeitung-SoSe-2011/Folien/Signalverarbeitung-5.pdf

https://documentation.apple.com/en/finalcutpro/usermanual/index.html#chapter=C%26section=12%26tasks=true